

或许这不是你所期望的答案，  
但这是你提出的请求

(一个意外的血统)

康纳·麦克布赖德

2019年10月23日



# 内容

1 Haskell 奇趣	7
1.1 在 Haskell 中, () 到底是什么? .....	7
1.2 在 Haskell 中, () 意味着什么? .....	8
1.3 在 Haskell 中实现 lines 函数 .....	9
1.4 布尔表达式求值 (微妙的类型错误) .....	9
1.5 纯函数式数据结构用于文本编辑器 .....	10
1.6 在 Haskell 中, Cofree CoMonad 有哪些激励性示例? .....	12
1.7 在列表中查找元素的索引 .....	15
1.8 如何将这个 mergeWords 函数扩展到任意数量的字符串? .....	15
1.9 UndecidableInstances 导致非终止类型检查的示例 .....	16
1.10 为什么 3 和 x (x 被赋值为 3) 在 Haskell 中有不同的推导类型? ..	17
1.11 排名为 3 (或更高) 的多态性有什么应用案例? ...	18
1.12 为什么 Haskell 编译器不支持确定性内存管理? .....	19
1.13 ArrowLoop 如何工作? 另外, mfix 又是怎么回事? .....	19
1.14 在类型签名中, => 意味着什么? .....	20
1.15 Double 和浮动点数的含义? ...	21
1.16 Haskell 术语: 类型与数据类型的含义, 它们是同义词吗? ...	22
1.17 能否将冒泡排序表述为一个单元或半群? .....	23
1.18 这是一个正确实现的 mergesort 吗? .....	24
1.19 Haskell 类型系统的细微差别 (歧义) .....	26
1.20 理解 Haskell 类型歧义的一个案例 .....	27
1.21 为什么 Haskell 使用 -> 而不是 =? .....	27
1.22 如果类型参数的顺序错误, 是否可能将一个类型实例化为类的实例? .....	28
1.23 Flip 数据类型的 Functor 类型变量 .....	29
1.24 为什么 product [] 返回 1? .....	30
1.25 两个 Maybe 值的最小值 ....	30
1.26 Haskell 的 Foldable 和 Traversable 的等效物是否仅仅是 Cl ojure 中的序列? 31	31
1.27 如何在不多次遍历字符串的情况下跟踪字符串的多个属性? .....	32
1.28 检查二叉树是否为二叉搜索树 .....	33
1.29 在二叉树中查找值为 x 的叶子 .....	33
1.30 从列表中取出元素, 直到遇到重复元素 .....	34
1.31 RankNTypes 和 PolyKinds (量词交替问题) .....	35
1.32 如何用视图模式语法编写这个 case 表达式? .....	36
1.33 递归类型族 .....	36
1.34 确定一个值是否为 Haskell 中的函数 .....	38
1.35 自动 Functor 实例 (不是) .....	38
1.36 如何应用第一个有效的部分函数在 Haske ll 中? .....	39
1.37 将普通列表与嵌套列表 “配对” .....	39
1.38 批量累加 .....	41
1.39 在 Phantom 类型上使用 Fu nctor .....	42

1.40 在 Haskell 中创建一个（带存储的）解释器 .....	43
1.41 存在类型包装器的必要性 .....	45
1.42 类型级函数中的非线性模式 .....	46
1.43 玫瑰树的初始代数 .....	46
2 模式匹配 .....	49
2.1 用于类型检查类 ML 模式匹配的算法? .....	49
2.2 Haskell 模式匹配 .....	50
2.3 复杂模式匹配 .....	51
2.4 如何返回我进入之前的一个元素? .....	52
2.5 Buzzard Bazooka Zoom .....	52
2.6 为什么在模式匹配中不允许 ++? .....	54
3 递归 .....	55
3.1 什么是参数形态 (paramorphisms)? .....	55
3.2 为什么在 Haskell 中可以用 foldl 反转列表, 但不能用 foldr? .....	56
3.3 fold 可以用来创建无限列表吗? .....	59
3.4 如何为一个为通用递归方案构建的数据类型提供 Functor 实例? .....	59
3.5 Haskell 中是否存在 (进行项变换的) 态射? .....	61
3.6 这个斐波那契数列函数是递归的吗? .....	62
3.7 能有人解释一下这个惰性的斐波那契解法吗? .....	63
3.8 固定点上的单子折叠 (Monoidal folds) .....	64
3.9 在求值列表元素时创建列表 .....	65
3.10 GADT 的函数 .....	66
4 应用函子 .....	69
4.1 在哪里可以找到适用于函子的编程练习? .....	69
4.2 N 叉树遍历 .....	71
4.2.1 第一次尝试: 繁重的工作 .....	71
4.2.2 第二次尝试: 编号和线程化 .....	72
4.2.3 第三次尝试: 类型导向编号 .....	74
4.2.4 最终结果 .....	75
4.3 函数的部分应用和柯里化, 如何写出比多个 map 更好的代码? .....	76
4.4 将 monad 转换为 applicative .....	77
4.5 Applicatives 可以组合, monads 不可以 .....	78
4.6 分离 Functor、Applicative 和 Monad 的示例 .....	79
4.7 Parsec: Applicatives 与 Monads .....	80
4.8 将 do 语法重构为 applicative 风格 .....	81
4.9 用默认值进行 zip, 而不是丢弃值? .....	82
4.10 使用 Haskell 中的 zipWith3 实现 sum3 .....	82
4.11 'Const' applicative functor 有什么用处? .....	83
4.12 免费 monad 的 applicative 实现 .....	83
4.13 适用于 monad 的示例, 其中 Applicative 部分可以比 Monad 部分更好地优化 .....	84
4.14 “ap” 实现对于 monads 有多随意? .....	85
4.15 没有 functor 的 applicative (用于数组) .....	85
4.16 这个简单的 Haskell 函数已经有了一个众所周知的名字吗? (strength) .....	86
4.17 孩子们都没问题 .....	87
4.18 为什么 ((,) r) 是一个不是 Applicative 的 Functor? .....	87
4.19 Applicative 重写 (用于 reader) .....	88
4.20 序列化对角化 .....	89
4.21 中缀运算符的 Applicative 风格? .....	89
4.22 Applicative 中的 Monoid 在哪里? .....	89

4.23 来自么半群的 Applicative (包括 min 和 max)	93
5 单子	95
5.1 为什么我们使用单子函数 $a \rightarrow m b$	95
5.2 使用 Join 而非 Bind 的单子	96
5.3 在列表单子中使用 return 与不使用 return	97
5.4 展示单子不能组合的示例	98
5.5 Pause 单子	98
5.6 Haskell 单子 return 任意数据类型	100
5.7 我是否应该避免使用 Monad 的 fail?	100
5.8 为什么 Kleisli 不是 Monoid 的一个实例?	101
5.9 在提示符下使用单子?	102
5.10 这是一个使用 liftM 的场景吗?	102
5.11 Zappy 共列表是否构成一个单子 not	104
5.12 Haskell io-streams 与 forever 不向 stdout 产生任何输出	104
6 类型的微分演算	105
6.1 在列表中查找某个元素的前驱元素	105
6.2 拆分列表	106
6.3 将 nub 表示为列表推导式	108
6.4 如何让二叉树拉链成为 Comonad 的一个实例?	109
6.5 Data.Void 中的 absurd 函数有什么用?	115
6.6 为 n 维网格编写 cojoin 或 cobind	117
6.6.1 列表中的游标	117
6.6.2 组合游标, 转置游标?	118
6.6.3 Hancock 的张量积	120
6.6.4 张量积的 InContext	121
6.6.5 Naperian 函子	122
6.7 通用的拉链 Comonad	122
6.8 Traversable 与拉链: 必要性与充分性	131
6.9 如何以地道的方式编写这个 (有趣的 filter) 函数?	131
6.10 计算一个依赖于所有前序项的列表项	132
6.11 合理的 Comonad 实现 (针对非空列表)	137
6.12 可表示 (或 Naperian) 函子	140
6.13 将 Trie 作为 Naperian 函子; 通过其导数进行匹配	143
7 依赖类型的 Haskell	149
7.1 最适合“现实世界”编程的依赖类型语言?	149
7.2 为什么不使用依赖类型?	149
7.3 给初学者的 Haskell 中的简单依赖类型示例。它们在 Haskell 的实践中有什么用? 我为什么要关心依赖类型?	154
7.4 Haskell 单例 (singletons): 使用 SNat 我们获得了什么?	155
7.5 对数据种类提升 (data kind promotion) 加以限制的动机	156
7.6 什么是索引单子 (indexed monad)?	157
7.7 索引集合上函子的固定点	160
7.8 为什么类型系统拒绝了我看似有效的程序?	161
7.9 是否可以在 Haskell 中编程并检查不变量?	164
7.10 为什么 typecase 是一件坏事?	166
7.11 为什么我不能对类型族进行模式匹配?	166
7.12 正整数类型	167
7.13 测试一个值是否匹配某个构造子	168
7.14 Haskell 与 Idris 的区别: 运行时/编译时在类型宇宙中的反映	169
7.15 为什么 GADT/存在数据构造子不能用于惰性模式?	170

7.16 GADTs 能否用于证明 GHC 中的类型不等式? .....	171	7.17 实现一个用于长度索引列表的拉链 .....	171
7.18 模整数的单元 .....	174	7.19 是否存在看起来不像容器的非平凡 Foldable 或 Traversable 实例? .....	176
7.20 GHC 的类型族是系统 F-omega 的一个例子吗? .....	177	7.21 如何为具有非 * 型态幻像类型参数的 GADT 推导 Eq? .....	178
7.22 Haskell 中的所有类型类是否都有一个范畴理论的类似物? .....	179	7.23 Haskell 类型解析中的类型类 (生成器、共单子) .....	180
7.24 证明类型级析取的幂等性 .....	181	7.25 递归定义的实例和约束 .....	182
7.26 如何获取依赖类型区间的长度? .....	184	7.27 如何使卡塔摩尔菲斯与参数化/索引类型一起工作? .....	185
7.28 限制签名中的构造函数 .....	186	7.29 类似 Either 的三种情况求和类型的标准名称是什么? .....	187
7.30 如何为异构集合在 GADT 形式的 AST 中指定类型? .....	188	7.31 类型线程化的异构列表和使用类型族的默认值 (?) .....	189
7.32 通过 DataKinds 提升到 $* \rightarrow A$ 的构造函数 .....	189	7.33 “引理”函数的一般类型应如何理解? .....	189
8 类型论 193			
8.1 直觉主义类型论在组合逻辑中的对应物是什么? .....	193	8.2 Haskell 或 Agda 是否有等化子 (equalisers)? .....	198
8.3 为什么我们需要容器 (containers)? .....	200	8.4 Applicative/Monad 实例在多大程度上是唯一确定的? .....	204
8.5 观测类型论 (Observational Type Theory) 中的模式匹配 .....	206	8.6 OTT 中的可证明一致性 (coherence) .....	207
8.7 如何在 Coq 中解决包含无效类型等式的目标? .....	208	8.8 是否可以在构造演算 (calculus of constructions) 中表达无标签平衡二叉树的类型? .....	210
8.9 在 Coq/Proof General 中进行类似 Agda 的编程? .....	211		

# 第1章

## Haskell 奇趣

### 1.1 Haskell 中的 () 究竟是什么？

太长不看：() 并不会给每一种类型都添加一个“null”值，当然不是；() 是一个“无趣的”值，且属于它自己的类型：()。

让我稍微退后一步，先处理一个常见的混淆来源。在学习 Haskell 时，一个关键的点是理解它的 *expression* 语言和 *type* 语言之间的区别。你可能知道这两者是分开的。但这也意味着相同的符号可以在两者中都使用，这正是这里发生的情况。有一些简单的文本提示可以告诉你正在查看的是哪种语言。你无需解析整个语言就能发现这些提示。

Haskell 模块的顶层默认位于表达式语言中。你通过编写表达式之间的方程来定义函数。但是当你在表达式语言中看到  $foo :: bar$  时，这意味着  $foo$  是一个表达式， $bar$  是它的类型。所以当你看到  $() :: ()$  时，你看到的是一个语句，它将表达式语言中的 () 和类型语言中的 () 联系起来。这两个 () 符号意味着不同的东西，因为它们不在同一种语言中。这种重复通常会让初学者感到困惑，直到表达式/类型语言的分离深入他们的潜意识，届时它将变得具有帮助性并且便于记忆。

关键字 `data` 引入了一种新的数据类型声明，涉及对表达式语言和类型语言的精妙混合：它首先说明新类型是什么，其次说明它的取值是什么。

在这样的声明中，类型构造器 *TyCon* 被添加到类型语言中，*ValCon* 值构造器被添加到表达式语言中（以及其模式子语言）。在数据声明中，作为 *ValCon* 的参数位置的事物告诉你当该 *ValCon* 在表达式中使用，赋予参数的类型。例如，

数据 树  $a = \text{叶子} \mid \text{节点 (树 } a) a \text{ (树 } a)$

声明了一个类型构造器 *Tree*，用于表示在节点处存储元素的二叉树类型，其值由值构造器 *Leaf* 和 *Node* 给出。我喜欢把类型构造器 (*Tree*) 标为蓝色，把值构造器 (*Leaf*、*Node*) 标为红色。在表达式中不应该出现蓝色，而在类型中（除非你使用高级特性）不应该出现红色。内建类型 *Bool* 可以被声明，

数据 布尔 = 真  $\mid$  假

将蓝色的 *Bool* 加入类型语言，并将红色的 *True* 和 *False* 加入表达式语言。遗憾的是，我的 Markdown 功夫不足以在这篇文章中添加颜色，所以你能在脑海中学会自己给它们加上颜色。

“unit”类型使用 () 作为特殊符号，但其工作方式如同已声明

数据 () = () -- 左侧 () 为蓝色；右侧 () 为红色

这意味着，名义上为蓝色的 `()` 是类型语言中的一种类型构造器，而名义上为红色的 `()` 是表达式语言中的一种值构造器，并且确实有 `() :: ()`。[这并不是这种双关的唯一例子。更大元组的类型遵循同样的模式：对偶（pair）语法仿佛由如下给出

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

将 `()` 添加到类型和表达式语言中。但我偏题了。

因此，类型 `()`，通常读作“Unit”，是一种只包含一个值得一提的值的类型：这个值在表达式语言中写作 `()`，有时读作“void”。只有一个值的类型并不怎么有趣。类型 `()` 的值提供零比特的信息：你已经知道它必然是什么。因此，尽管类型 `()` 本身并没有任何表明副作用的特殊之处，它却经常作为单子类型中的值组件出现。单子操作往往具有如下形式的类型

其中返回类型是一个类型应用：函数告诉你可能有哪些效果，而参数告诉你该操作会产生什么类型的值。例如

```
放置 :: s -> 状态 s ()
```

这是被解读为（因为应用关联到左侧 [“就像我们在六十年代所做的那样”，Roger Hindley]）

```
放置 :: s -> (状态 s) ()
```

它有一个值输入类型 `s`、效果单子 `State s`，以及值输出类型 `()`。当你看到 `()` 作为值输出类型时，这只是意味着“这个操作仅用于它的 *effect*；返回的值并不重要”。同样地

```
putStr :: String -> IO ()
```

将字符串输出到 `stdout`，但不返回任何令人兴奋的内容。

`()` 类型也可作为类似容器结构的元素类型使用，在这种情况下，它表示数据仅由一个 *shape* 构成，没有任何有意义的负载。例如，如果 `Tree` 如上所声明，那么 `Tree ()` 就是二叉树形状的类型，在节点处不存储任何有意义的内容。类似地，`[]` 是由乏味元素组成的列表类型，如果列表的元素中没有任何值得关注的内容，那么它所提供的唯一信息就是其长度。

总结来说，`()` 是一种类型。它的一个值，`()` 恰好有相同的名称，但这没关系，因为类型和表达式语言是分开的。拥有一个表示“没有信息”的类型是有用的，因为在上下文中（例如单子或容器的上下文），它告诉你只有上下文才是有意义的。

## 1.2 `()` 在 Haskell 中是什么意思？

`()` 表示“无聊”。它指的是只包含一个事物的那种无聊的类型，本身也很无聊。将这种无聊类型的一个元素与另一个进行比较，无法获得任何有趣的东西，因为即使你给予它任何注意力，也无法从这种无聊类型的元素中学到任何东西。

它与空类型非常不同；在 Haskell 中，这个空类型被称为 `Void`（这个名字是别人起的，我真希望他们能选一个更好的名字，比如我建议的那个）。空类型非常令人兴奋，因为如果有人真的给了你一个属于它的值，你就知道自己已经死了、身在天堂，而且你想要的任何东西都是你的。

但如果有人给你一个括号中的值，不要兴奋。直接把它丢掉。

有时候，把由“元素类型”参数化的类型构造子拿来，用 `()` 填充这个参数，会很有趣。这样你就能看清类型构造子本身所固有的信息，而不是来自元素的信息。例如，`Maybe ()` 是 `Bool` 的一种版本，具有 `Just ()` 或

什么也没有。而且，`[]` 相当于（可能是无限的）自然数：你拥有的唯一信息就是长度。

所以，`()` 意味着“无聊”，但它通常是一个线索，表示某个地方正在发生有趣的事情。

## 1.3 在 Haskell 中实现函数 *lines*

假设你已经知道除了输入的第一个字符之外的所有行的结果。那么如何将第一个字符加到这个结果上呢？

```
charon :: Char -> [[Char]] -> [[Char]]
charon ' \n' css = [] : css -- 开始时换行，插入空行
charon c [] = [[c]] -- 最后一个字符单独一行
charon c (cs : css) = (c : cs) : css -- 否则将字符放在第一行
```

谜团解开之后，

```
行 = foldr charon []
```

多年来，我一直让学生们用拳头砸家具，并高喊“*what*你和*empty list*有关系吗？*what*你和*x cons xs*有关系吗？”。有时这管用。

## 1.4 布尔表达式评估（细微类型错误）

你的怀疑——“我认为数据 `BExp` 声明本身存在问题”——是正确的。你写的内容并不意味着我猜想我希望它表达的含义。错误出现在最右端（错误往往如此），所以我不得不滚动才能找到它。我们多用一些纵向空间，来看一看。

```
数据 BExp = 等于 AExp
AExp | 小于 AExp AExp
p | 大于 AExp AExp |
小于等于 AExp AExp |
大于等于 AExp AExp |
与 BExp BExp | 或 BExp
p BExp | 布尔
```

而最后一个才是大问题。它更难发现，因为尽管你告诉 *us*，“我希望类型是 `:: BExp -> Bool`”，但你没有告诉编译器。如果你做了该做的事，通过写明类型签名来表达你的意图，错误报告可能会更有帮助。你的程序开始

```
evalBExp 真 = 真
```

并且这足以说服类型检查器，表明预期的类型是

```
evalBExp :: Bool -> Bool
```

因为 `True :: Bool`。当第3行出现时

```
evalBExp (Eq a1 a2) = evalAExp (a1) == evalAExp (a2)
```

突然它开始怀疑为什么你给 `evalBExp` 一个 `BExp` 而不是一个 `Bool`。现在，我怀疑你有这样的印象，即你在 `BExp` 中的最终子句

| 布尔

创建了 `True :: BExp` 和 `False :: BExp`，但它实际上并不是这么做的。相反，你会发现你有一个零参数数据构造器 `Bool :: BExp`，它的名字与数据类型 `Bool` 相同，但存在于一个完全独立的命名空间中。我相信你的意图是将 `Bool` 的值隐式地嵌入 `BExp` 中，但 `Haskell` 不允许如此微妙的子类型化。为了实现预期的效果，你需要一个显式打包 `Bool` 的构造器，所以试试这样做：

```
数据 BExp = ... |
BVal 布尔
```

和

```
evalBExp :: BExp -> 布尔值 evalB
Exp (BVal b) = b
...
```

更接近你的计划。

当然，您可以自由地使用名称 `Bool` 作为 `BVal` 构造函数，因此可以写作

```
数据 BExp
= ... | 布尔 布
尔
```

其中第一个 `Bool` 是数据构造函数，第二个是类型构造函数，但我会觉得这个选择令人困惑。

## 1.5 纯粹函数式数据结构用于文本编辑器

我不知道这个建议是否对“好”的复杂定义来说是“好”的，但它简单且有趣。我经常设定一个练习，让学生用 `Haskell` 编写文本编辑器的核心，并与我提供的渲染代码连接。数据模型如下。

首先，我定义了一个包含 `x` 元素的列表中作为游标的含义，其中游标处可用的信息具有某种类型 `m`。（其中 `x` 将被证明是 `Char` 或 `String`。）

```
type Cursor x m = (Bwd x, m, [x])
```

这个 `Bwd` 东西其实就是倒序的“`snoc-lists`”。我想保持强烈的空间直觉，所以我在代码中反转操作，而不是在脑海中反转。这个想法是，离光标最近的东西是最容易访问的。这就是 `The Zipper` 的精神。

```
数据 Bwd x = B0 | Bwd x :< x 派生 (Show, Eq)
```

我提供了一个免费的单例类型，作为光标的可读标记。...

```
数据 这里 = 这里 推导 Show
```

...因此我可以说在字符串中的某个位置是什么。

```
类型 StringCursor = 游标 字符 在这里
```

现在，为了表示一个包含多行的缓冲区，我们需要在光标所在行的上下方分别有字符串，并在中间使用一个 `StringCursor`，表示我们当前正在编辑的行。

```
类型 TextCursor = 光标 字符串 字符串光标
```



## 1.6 在 Haskell 中, Cofree 共单子有哪些动机性的示例?

让我们简单回顾一下 Cofree 数据类型的定义。

数据 Cofree f a = a :< f (Cofree f a)

这至少足以诊断示例中的问题。当你写道

```
1 :< [2, 3]
```

你犯了一个小错误, 这个错误被报告得比实际更微妙一些。在这里,  $f = []$  和  $a$  是某个数值, 因为  $1 :: a$ 。因此, 你需要

```
[2, 3] :: [Cofree [] a]
```

因此

```
2 :: Cofree [] a
```

如果 Cofree [] a 也是 Num 的一个实例, 那样 *could* 就没问题。然而, 你的定义因此获得了一个不太可能被满足的约束; 事实上, 当你 *use* 你的值时, 试图满足该约束的尝试会失败。

再次尝试使用

```
1 :< [2 :< [], 3 :< []]
```

你应该会更好运。

现在, 让我们看看我们得到了什么。从简单开始。Cofree f () 是什么? 特别地, Cofree [] () 是什么? 后者与 [] 的固定点同构: 每个节点是子树列表的树结构, 也被称为“无标签的玫瑰树”。例如,

```
() :< [ () :< [ () :< [],
                , () :< []
              ],
        , () :< []
      ]
```

同样地, Cofree Maybe () 或多或少是 Maybe 的不动点: 自然数的一份拷贝, 因为 Maybe 要么给我们零, 要么给我们一个可以插入子树的位置。

```
zero :: Cofree Maybe ()
zero = () :< Nothing
succ :: Cofree
Maybe () -> Cofree Maybe ()
succ n = () :< Just n
```

一个重要的琐碎案例是 Cofree (Const y) (), 它是  $y$  的副本。Const y 函子为子树提供 *no* 个位置。

```
pack :: y -> Cofree (Const y) ()
pack y = () :< Const y
```

接下来, 让我们处理另一个参数。它告诉你为每个节点附加什么标签。更具提示性的重命名参数

```
数据 Cofree nodeOf label = label :< nodeOf (Cofree nodeOf label)
```

## 1.6. Haskell 中 Cofree Comonad 的一些动机性示例是什么? 13

当我们标记 (Const y) 示例时, 我们得到 *pairs*

```
pair :: x -> y -> Cofree (Const y) x
pair x y = x :< Const y
```

当我们 将标签附加到我们数字的节点上, 我们得到 *nonempty* 列表

```
one :: x -> Cofree Maybe x
one = x :< Nothing
construct :: x -> Cofree Maybe x
construct x xs = x :< Just xs
```

而对于列表, 我们得到 *labelled* 玫瑰树。

```
0 :< [ 1 :< [ 3 :< []
      , 4 :< []
      ]
    , 2 :< []
    ]
```

这些结构始终是“非空”的, 因为至少存在一个顶节点, 即使它没有子节点, 而且该节点总会有一个标签。提取操作会返回顶节点的标签。

```
extract :: Cofree f a -> a
extract (a :< _) = a
```

也就是说, 提取操作丢弃了顶部标签的 *context*。现在, 重复操作 *decorates* 每个标签与 *its own* 上下文。

```
重复 :: Cofree f a -> Cofree f (Cofree f a)
重复 a :< fca = (a :< fca) :< fmap 重复 fca -- f 的 fma
p
```

我们可以通过遍历整棵树为 Cofree f 获得一个 Functor 实例

```
fmap :: (a -> b) -> Cofree f a -> Cofree f b
fmap g (a :< fca) = g a :< fmap (fmap g) fca --
^^^^ ^^^^^ f 的 fmap ||| -- (Cofree f) 的 fmap, 递归使用
```

很容易看出

```
fmap extract . duplicate = id
```

因为 duplicate 用上下文装饰每个节点, 那么 fmap extract 就丢弃了这些装饰。

注意, fmap 在计算输出的标签时只能查看输入的标签。假设我们希望根据每个输入标签 *in its context* 来计算输出标签呢? 例如, 给定一棵未标注的树, 我们可能想给每个节点标注其整个子树的大小。得益于 Cofree f 的 Foldable 实例, 我们应该能够对节点进行计数。

```
长度 :: 可折叠 f => Cofree f a -> 整数
```

这意味着

```
fmap length . duplicate :: Cofree f a -> Cofree f Int
```

共单子的关键思想是它们捕获“具有某种上下文的事物”，并且它们允许你在任何地方应用依赖上下文的映射。

扩展 :: 共单子  $c \Rightarrow (c\ a \rightarrow b) \rightarrow c\ a \rightarrow c\ b$  扩展  $f = \text{fmap } f$  -- 上下文相关的映射到处 . --  
在重复之后 -- 用其上下文装饰一切

更直接地定义 `extend` 可以省去重复的麻烦（尽管这实际上只是共享）。

```
extend :: (Cofree f a -> b) -> Cofree f a -> Cofree f b
extend g ca@(_:< fca) = g c
a:< fmap (extend g) fca
```

并且你可以通过采取来取回重复项

复制 = 扩展 `id` -- 输出标签是其上下文中的输入标签

此外，如果你选择对每个上下文中的标签执行 `extract`，那么你只是把每个标签放回它原来的位置：

扩展提取 = `id`

这些 “对上下文中标签的操作” 被称为 “co-Kleisli arrows”，

```
g :: c a -> b
```

扩展的任务是将 co-Kleisli 箭头解释为对整个结构的函数。提取操作是身份 co-Kleisli 箭头，它被扩展解释为身份函数。当然，还有 co-Kleisli 合成。

```
(=<=) :: 余单子 c => (c s -> t) -> (c r -> s) -> (c r -> t)
(g =<= h) = g . 扩展 h
```

而共单子定律确保 `=<=` 是结合的并且吸收 `extract`，从而得到 co-Kleisli 范畴。此外，我们有

```
扩展 (g =<= h) = 扩展 g . 扩展 h
```

因此，扩展是从 co-Kleisli 范畴到集合和函数的 *functor*（在范畴意义上的）。这些定律对于 `Cofree` 来说并不难验证，因为它们源自于节点形状的函子定律。

现在，一种有用的方式是将 `cofree comonad` 中的结构看作一种“游戏服务器”。一种结构

```
a:< fca
```

表示博弈的状态。博弈中的一步要么是“停止”，在这种情况下你得到 `a`；要么是“继续”，即选择 `fca` 的一个子树。例如，考虑

余自由代数（`(->)` 移动）奖

一个客户端必须停止，或者通过给出一个动作继续：这是一个 *list* 的动作。游戏按如下方式进行：

```
play :: [move] -> Cofree ((->) move) prize -> prize
play [] (prize:< _) = prize
play (m:ms) (_:< f) = play ms (f m)
```

也许一个 `move` 是一个 `Char`，而奖品是解析字符序列的结果。

如果你盯得足够久，你会发现 `[move]` 实际上是 `Free ((), move) ()` 的一个版本。`Free` 单子表示客户端策略。函子 `(((), move)` 相当于一个只有“发送一个 `move`”这一条命令的命令接口。函子 `((->) move)` 则是相应的结构——“对发送一个 `move` 进行响应”。

一些函子可以被看作是刻画了一种命令接口；此类函子的自由单子表示会发出命令的程序；该函子将有一个“对偶”，它表示如何响应命令；该对偶的余自由余单子是一个一般性的环境概念，在其中可以运行会发出命令的程序，其中标签说明当程序停止并返回一个值时该做什么，而子结构说明当程序发出命令时如何继续运行程序。

例如，

数据 通信 `x = 发送 字符 x | 接收 (字符 -> x)`

描述允许发送或接收字符。其对偶是

数据 响应者 `x = 响应 {ifSend :: 字符 -> x, ifReceive :: (字符, x)}`

作为练习，看看你是否能够实现该交互

闲聊 :: 自由通信 `x ->` 余自由响应器 `y -> (x, y)`

## 1.7 在列表中查找事物的索引

我会使用 `zip` 和列表推导式。

`indicesOf :: Eq a => a -> [a] -> [Int]` `indicesOf a` 作为 `[i | (b, i) <- zip 作为 [0..], b == a]`

使用`[0..]`进行压缩是一种标准方式，可以为每个元素标注一个索引，然后它就是一个简单的查询。

## 1.8 我如何将这个 `mergeWords` 函数扩展到任意数量的字符串？

这个难题实际上是将一个 `list` 的单词，每次一个字符，合并成带有换行符的行。

`mergeWords :: [String] -> 字符串`

我们需要取一个类似这样的列表

```
[ "你好", "
吉姆", "很
高兴", "今
天" ]
```

并将其重新排列为给定位置的事项列表

```
[ "hjnd", "e
iia", "lmcy
", "le", "o"
]
```

这正是库函数 `transpose` 的功能。

然后我们需要生成一个单一的字符串，把这些当作各行，并用换行符分隔。这正是 `unlines` 所做的事情。

所以

合并单词 = `unlines . transpose`

我们完成了。

## 1.9 UndecidableInstances 导致类型检查不终止的示例

这篇来自 HQ 的论文中有一个经典、且略显令人担忧的例子（涉及与函数依赖的相互作用）。

类 `Mul a b c | a b -> c` 其中 `mul :: a -> b -> c` 实例 `Mul a b`  
`c => Mul a [b] [c]` 其中 `mul a = map (mul a)`

`f b x y = 如果 b 那么 mul x [y] 否则 y`

我们需要 `mul x [y]` 使其与 `y` 具有相同的类型，因此，取 `x :: x` 和 `y :: y`，我们需要实例 `Mul x [y] y`

根据给定的实例，我们可以得到。当然，我们必须取 `y [z]`，其中 `z` 满足

实例 `Mul x y z`

即

实例 `Mul x [z] z`

而我们陷入了一个循环。

这令人不安，因为那个 `Mul` 实例看起来其递归在结构上是 *reducing*，不像 Petr 的回答中那个明显病态的实例。然而它却让 GHC 陷入循环（触发了 `boredom threshold` 以避免挂起）。

问题在于，正如我确信自己在某处某时提到过的那样，尽管 `z` 在函数上依赖于 `y`，仍然进行了 `y [z]` 的统一。如果我们对这种函数依赖使用函数式记法，我们或许会看到该约束表明 `y Mul x [y]`，并因此因违反出现检查而拒绝这种替换。

出于好奇，我想试一试，

```
class Mul' a b where type MulT a b mul' :: a -> b -
> MulT a b instance Mul' a b => Mul' a [b] where t
ype MulT a [b] = [MulT a b] mul' a = map (mul' a)
g b x y = if b then mul' x [y] else y
```

在启用 `UndecidableInstances` 时，循环需要相当长的时间才会超时。禁用 `UndecidableInstances` 时，该实例仍然被接受，类型检查器仍然会陷入循环，但触发截断要快得多。

所以……这真是个有趣的世界。

## 1.10 为什么在 Haskell 中，3 和 x（被赋值为 3）会被推断出不同的类型？

这里还有一个因素，在 [acftoltzer](#) 提供的一些链接中提到过，但可能值得在这里明确指出。你正在遇到单态限制的影响。当你说

设  $x = 5$

你对一个 *variable* 进行了 *top-level* 定义。MR 坚持认为，这样的定义在没有类型签名相伴时，应当通过为未解析的类型变量选择（希望是）合适的默认实例，将其专门化为单态值。相比之下，当你使用 `:t` 来请求推断类型时，不会施加这样的限制或默认化。所以

```
> :t 3 3 :: (Num t) => t
```

因为 3 确实是重载的：它被任何数值类型所接受。默认化规则选择 `Integer` 作为默认的数值类型，因此

```
> let x = 3 > :t x
x :: Integer
```

但现在我们关闭 MR。

```
> :set -XNoMonomorphismRestriction > let y = 3
> :t y y :: (Num t) => t
```

没有 MR，这个定义可以说是能有多多态就有多多态，和 3 一样重载。只是确认一下……

```
> :t y * (2.5 :: Float) y * (2.5 :: Float) ::
Float > :t y * (3 :: Int) y * (3 :: Int) :: Int
```

注意，多态的  $y = 3$  在这些用法中会根据相关 `Num` 实例所提供的 `fromInteger` 方法而被不同地特化。也就是说， $y$  并不与 3 的某个特定表示相关联，而是与一种用于构造 3 的表示的方案相关。若以朴素方式编译，这就是导致性能缓慢的配方，一些人将其作为 MR 的动机。

我在关于单态限制是较小的邪恶还是较大的邪恶的辩论中（在本地假装）持中立态度。我总是为顶级定义写类型签名，这样就没有关于我想要实现的目标的歧义，而 MR 与此无关。

在尝试学习类型系统如何工作时，将类型推断的各个方面分开来看是非常有用的，

1. “遵循计划”，将多态定义专门化到特定用例：这是一个相当健壮的约束求解问题，需要通过回溯链进行基本的统一和实例解析；以及
2. “猜测计划”，通过类型泛化为没有类型签名的定义分配一个多态类型方案：这相当脆弱，而且越是超出基本的 Hindley–Milner 规范，引入类型类、高阶多态、GADT，事情就会变得越发怪异。

了解第一个如何工作确实是好事，并且理解为什么第二个很困难也很重要。类型推断中的许多奇怪之处都与第二个有关，并且与像单态限制这样的启发式方法相关，试图在面对歧义时提供有用的默认行为。

## 1.11 三阶（或更高阶）多态的使用场景？

我也许能帮上忙，尽管这种野兽不可避免地有点复杂。下面是一种我在开发具有良好作用域、带有绑定和 de Bruijn 索引的语法时有时会使用的模式，打包呈现。

```
``markdownmkRenSub :: forall v t x y. -- 变量由 v 表示，术语由 t 表示 (forall x. v x -> t x) -> -- 如何将变量嵌入到术语
中 (forall x. v x -> v (Maybe x)) -> -- 如何平移变量 (forall i x y. -- 对于事物 i，如何遍历术语... (forall z. v z -> i z) -> -- 如何
从变量生成事物 (forall z. i z -> t z) -> -- 如何从事物生成术语 (forall z. i z -> i (Maybe z)) -> -- 如何削弱事物 (v x -> i y
) -> -- ...将变量转换为事物 t x -> t y) -> -- 它们出现的地方 ((v x -> v y) -> t x -> t y, (v x -> t y) -> t x -> t y) -- 获取重命
名和替换``
```

```
mkRenSub var weak mangle = (ren, sub) 其中 ren = mangle id var weak -- 将对象视为变量以获得重命名替换 sub = m
angle var id (ren weak) -- 将对象视为项以获得替换
```

通常我会用类型类来掩盖最丑陋的细节，但如果你把这些字典拆开，就会发现其实是这样。

关键在于，mangle 是一种秩为 2 的操作，它接受一种配备了适当操作的 thingy 概念，而这些操作在其所作用的变量集合上是多态的：将变量映射为 thingy 的操作会被转化为项变换器。整个过程展示了如何使用 mangle 来生成重命名和替换。

这是该模式的一个具体实例：

```
数据 Id x = Id x
```

```
数据 Tm x
  = Var (Id x) | App (Tm x) (
    Tm x) | Lam (Tm (Maybe x))
  )
```

```
tmMangle :: forall i x y.
  (forall z. Id z -> i z) -> forall z. i z -> Tm z -> forall z. i z -> i (Maybe z) -> (Id x -> i y) -> Tm x
-> Tm y tmMangle v t w f (Var i) = t (f i) tmMangle v t w f (App m n) = App (tmMangle v t w f m) (tmMangl
e v t w f n) tmMangle v t w f (Lam m) = Lam (tmMangle v t w g m) 其中 g (Id Nothing) = v (Id Nothing) g (I
d (Just x)) = w (f (Id x))
```

```
subst :: (Id x -> Tm y) -> Tm x -> Tm y
subst = snd (mkRenSub Var (\ (Id x) -> Id (Just x)) tmMangle)
```

我们只实现一次术语遍历，但以一种非常通用的方式；随后通过部署 mkRenSub 模式（该模式以两种不同方式使用这一通用遍历）即可获得替换。作为另一个例子，考虑类型算子之间的多态操作

```
type (f :-> g) = forall x. f x -> g x
```

一个 `IMonad`（索引单子）是一些 `m :: (* -> *) -> * -> *`，配备了多态运算符

```
ireturn :: 对于所有 p. p -> m p
iextend :: 对于所有 p q. (p -> m q) ->
m p -> m q
```

因此那些操作是二阶的。现在，任何由任意索引单子参数化的操作都是三阶的。所以，例如，构造通常的单子复合，

```
compose :: forall m p q r. IMonad m => (q -> m r) -> (p -> m q) -> p -> m r
compose qr pq = iextend qr . pq
```

依赖于等级 3 量化，一旦你解开 `IMonad` 的定义。

故事的寓意：一旦你在多态/索引概念上进行更高阶编程，你的有用工具字典将变为秩 2，而你的通用程序将变为秩 3。当然，这是一种可能再次发生的升级。

## 1.12 为什么 Haskell 编译器不支持确定性的内存管理？

询问函数式编程语言是否可以通过跟踪使用情况来减少 GC 是合理的。尽管判断某些数据是否可以安全丢弃的一般性问题是不可判定的（由于条件分支），但在静态分析上投入更多努力，找到更多直接释放内存的机会，显然是可行的。

值得关注 Martin Hofmann 及其团队在 Mobile Resource Guarantees 项目中的工作，他们将类型导向的内存（去/再）分配作为一个重要主题。不过，使他们的方案得以奏效的关键在于 Haskell 的类型系统中所缺乏的一样东西——线性性。如果你知道一个函数的输入数据与计算其余部分是 *secret* 的，那么你就可以重新分配它们所占用的内存。MRG 的工作尤其出色，因为它在一种类型的释放与另一种类型的分配之间管理了一种现实的交换率，而在纯函数式的外表之下，这最终会转化为传统的指针操作。事实上，许多精巧而节俭的可变算法（例如指针反转遍历、覆盖尾指针的构造等）都可以利用这些技术被塑造成纯函数式的形式（并且还能检查出讨厌的错误）。

实际上，资源的线性类型化为一种可能有助于减少 GC 的使用分析提供了一种保守但可机械检查的近似。接下来有趣的问题包括，如何将这种处理方式与通常的持久性方案干净地结合起来（刻意的副词选择）。在我看来，许多中间数据结构在递归计算中都有一个最初的单线程阶段，随后在计算结束时要么被共享，要么被丢弃。或许可以减少此类过程所产生的垃圾。

TL;DR 有一些好的类型化方法可以进行使用分析，从而减少 GC，但 Haskell 目前的类型信息不适合于此目的。

## 1.13 ArrowLoop 如何工作？另外，mfix 呢？

在这段代码中，关键部分是 `rec` 块中的 `delay 0` 箭头。为了理解它的工作方式，把值看作随时间变化，并将时间切分成若干片段会很有帮助。我把这些片段看作“天”。`rec` 块解释了每天的计算是如何进行的。它是按 *value* 组织的，而不是按 *causal order*，但只要我们足够小心，仍然可以追踪因果关系。至关重要的是，我们必须在没有任何来自 *types* 的帮助下，确保每天的工作依赖于过去而不是未来。一天的 `delay 0` 在这方面为我们争取了时间：它把输入信号向后推迟一天，并通过在第一天给出值 0 来处理这种情况。该 `delay` 的输入信号是“明天的 *next*”。

输出 <- returnA <- 如果重置则 0 否则下一步 下一步 <- 延迟 0 <- 输出+1

因此，从箭头及其输出来看，我们输出的是 *today's*，而下一步是 *tomorrow's*。从输入来看，我们依赖于 *today's* 的 *reset* 和 *next* 值。显然，我们无需时间旅行就能从这些输入得到这些输出。输出是今天的下一个数，除非我们将其重置为 0；到明天，下一个数就是今天输出的后继。因此，今天的 *next* 值来自昨天；如果没有昨天，则为 0。

在较低的层次上，整个设置之所以有效，是因为 Haskell 的懒计算。Haskell 采用按需驱动的策略进行计算，因此如果任务之间存在遵循因果关系的顺序，Haskell 会找到它。这里，延迟建立了这样的顺序。

不过要注意，Haskell 的类型系统在确保这种顺序存在方面几乎帮不上什么忙。你完全可以用循环去做彻头彻尾的胡闹之事！所以你的问题一点也不简单。每次你读或写这样的程序时，都确实需要思考“这到底怎么可能工作？”。你需要检查是否恰当地使用了 *delay*（或类似机制），以确保只有在信息能够被计算出来时才会去需求它。注意 *constructors*，尤其是 *(:)* 也可以像延迟一样起作用：在表面上给定整个列表的情况下计算列表的尾部并不罕见（但要小心只检查头部）。与命令式编程不同，惰性函数式风格允许你围绕事件顺序以外的概念来组织代码，但这种自由也要求对时间有更微妙的意识。

## 1.14 在类型签名中 $\Rightarrow$ 表示什么？

这是另一种看待它的方式。函数的一些参数是不可见的，另一些是可见的。类型输入  $\rightarrow$  输出告诉我们，作为参数需要一个可见的输入。类型 (Constraint)  $\Rightarrow$  输出告诉我们，期望有一些不可见的信息。它们不可互换，因为可见参数必须被写出来，而不可见参数则绝不能被写出来。不可见参数是供编译器自己去推断的（嗯，对我来说他听起来像个“他自己”），而且他坚持要把它们琢磨清楚：他拒绝仅仅被告知它们是什么！

实际上，这个 *tell* 示例的完整类型是

告诉 :: 对于  $(a :: *)$ 。  $(\text{Show } a) \Rightarrow [a] \rightarrow$  字符串

我所做的是明确指出这个变量出现的位置以及它是什么类型的东西。你也可以将其理解为一个“协议”：告诉所有满足需求的类型 *a* 的提议（显示 *a*）。

为了让一次对 *tell* 的使用有意义，它需要三样东西。其中两个是不可见的，一个是可见的。也就是说，当你使用 *tell* 时，你将可见的参数显式地给出，而编译器会尝试填补不可见的部分。让我们更慢一些地来梳理这个类型。

*tell* :: forall (a :: \*) . -- 要输出的元素类型（不可见）  $(\text{Show } a) \Rightarrow$  -- 如何从一个元素生成 String（不可见）  $[a] \rightarrow$  -- 要输出的元素列表（可见） String -- 通过显示所有元素生成的 String

所以，当你使用 *tell*，例如，

告诉 [True, False]

你只给出可见的参数：要说明的事物列表 [True, False]，而编译器会推断出不可见的参数。他知道 True 和 False 都是 Bool 类型的值，因此这意味着

[真, 假] :: [布尔值]

这就是编译器如何推断出 `tell` 的类型中的 `a` 必须是 `Bool`，从而使得 `[a] = [Bool]`

顺便提一下，关于 `[True, False] :: [Bool]`。在 `::` 的左侧，方括号 `[..]` 用于表示列表值。在 `::` 的右侧，方括号 `[..]` 用于表示列表的类型。它们在你眼中可能只是黑色的，背景是灰色的，但我的大脑将表示值的方括号标记为红色，将表示类型的方括号标记为蓝色。它们完全不同。我希望我能在在这个网站上使用颜色编码。我有些跑题了。

所以，现在，另一个看不见的参数必须满足这个 `(Show a)` 的要求，而我们现在知道它具体是 `(Show Bool)`，因为我们已经弄明白 `a` 是 `Bool`。我们称类型的这一部分为“约束”，但实际上它不仅仅是要求某个事实为真，更是要求某些有用的东西存在。这里要求的存在是一个函数

显示 :: 布尔值 -> 字符串

这就是在评估 `tell [True, False]` 的过程中，用于将单个元素 `True` 和 `False` 转换为字符串的函数。

标识符 `Show` 是一个 *type class* 的名称，而 `show` 是该类型类的方法。类型类规定了一组操作接口，这些操作必须为每个实例实现。该函数的隐式参数是一个记录（或“字典”），用于打包针对相关类型的这些操作的实现（这里是 `show` 的实现）。如果没有这些信息，编译后的代码将无法完成其工作；但我们不必手动编写这些信息，因为编译器（至少在这种情况下）可以在其已知的实例中进行搜索，并为当前任务填入正确的那个。

因此，我们不仅有不可见的类型参数（它们在编译期被推断出来，并在运行前被擦除），通过小写类型变量来标示，或更明确地通过 `forall blah ..`。我们还有类型类操作的不可见实现（它们在编译期被查找，用以提供关键的运行期信息）。因此，在 *between inferring and erasing types* 会发生一件非常重要的事情：当编译器仍然知道这些类型时，它会利用它们来确定运行时需要哪些不可见的实现，从而使我们无需亲自把它们写出来。

放大来看，`=>` 在类型文档中记录了我们的期望，即编译器将利用类型信息来指导生成我们无需费心编写的运行时代码。这是一个不错的小胜利。

类型系统黑客的潜在动机。不可见-可见的区分与可擦除-有用的区分存在于不同的位置，这是一些人尚未理解的信息。这是经典的Hindley-Milner观点，但事实是这些区分是正交的，越早让人们学会享受这一点越好。

## 1.15 双精度和浮动点的意义？

在 `ghci` 中以交互方式尝试的一件有用的事情是 `:info <something>` 命令，它有时可以告诉你一些有用的信息。

```
> :info 浮动类 Fractional a => 浮动 a 其中 pi :: a exp :: a -> a log :: a -> a sqrt :: a ->
a (**) :: a -> a -> a ----- 加载更多内容 -- 定义在 ‘GHC.Float’
```

实例 浮动 Float -- 定义于 'GHC.Float' 实例 浮动 Double -- 定义于 'GHC.Float'

这是什么意思？浮动是一个 *type class*。浮动点数有多种类型。实际上，标准有两种：Float 和 Double，其中 Double 提供的是 Float 的两倍精度。Floating a 中的 a 代表任何类型，而包含 sqrt 在内的大量操作是一个接口，任何该类的实例都必须实现这个接口。sqrt 出现在 Floating 的接口中意味着它只能并且始终只能用于 Floating 的实例。也就是说，对于你来说，它的类型是通过你所说的方式给定的。

平方根 :: 浮动类型 a => a -> a

=> 语法表示 *constraint*，这里将 Floating a 放置在其左侧。类型表示

对于任何类型 a，它是 Floating 的实例，给定类型 a 的输入，输出将具有类型 a。

您可以通过为类型填写任何可以满足约束条件 Floating a 的类型来专业化此类型，因此以下两个都为真

```
sqrt :: 浮动 -> 浮动
sqrt :: 双精度 -> 双精度
```

现在，Float 和 Double 由不同的位模式表示，因此每种情况下求平方根的计算机制不同。无需记住用于不同类型的不同版本的名称是非常方便的。“约束” Floating 实际上代表了接口中所有操作的实现记录（或 *dictionary*），适用于类型 a。sqrt 的类型实际表示的是

给定一个类型a和一个包含所有浮动操作实现的字典，我将接受一个a并返回一个a。

它通过从字典中提取相关的sqrt实现，并将其应用于给定的输入来工作。

因此，=> 表示一种具有不可见字典输入的函数类型，正如 -> 表示一种具有可见值输入的函数类型一样。当你使用该函数时，你不会（事实上也不能）写出这个字典：编译器会根据类型推断出来。当我们写下

```
sqrt :: 双精度 -> 双精度
```

我们指的是隐式地应用于 Floating Double 字典的通用 sqrt 函数。

## 1.16 Haskell 术语：type 与 data type 的含义，它们是同义词吗？

一种类型（在 Haskell 中）是一个语法元素，可以有意义地放在 :: 右侧，以对 :: 左侧的表达式进行分类。类型的每个语法组件本身由一种种类进行分类，其中表示类型（分类表达式）的种类是 \*。Some people are happy to use the word “type” to refer to any component of the type syntax, whether or not its kind allows it to classify expressions.

类型的语法可以通过各种声明形式扩展。

1. 一种类型同义词，例如，类型 `Foo x y z = [x] -> IO (y, z)`，添加了完全应用形式 `Foo x y z` 的类型组件，这些组件根据其定义方程以宏的方式展开。

2. 一个 `data` 声明，例如，`data Goo x y z = ThisGoo x | ThatGoo (Goo y z x)`，向类型语法中引入一个新的类型构造子符号 `Goo`，它用于构建对由数据构造子（此处为 `ThisGoo` 和 `ThatGoo`）生成的值进行分类的类型。3. 一个 `newtype` 声明，例如，`newtype Noo x y z = MkNoo (x, [y], z)`，复制一个已有类型，并在类型语法中与原始类型区分开来。

如果一个类型包含可以被其他类型组件替换的类型变量，那么它就是多态的：由多态类型分类的值可以被特化为类型变量的任何替换实例。例如，`append (++) :: [a] -> [a] -> [a]` 作用于元素具有相同类型的列表，但具体是什么类型都可以。具有多态类型的值通常被称为“多态值”。

有时，“数据类型”被用来简单地表示由数据声明引入的类型。从这个意义上讲，所有数据类型都是类型，但并非所有类型都是数据类型。不是数据类型的类型的例子包括 `IO ()` 和 `Int -> Int`。此外，`Int` 在 *this* 的意义上并不是数据类型：它是一个硬编码的原始类型。为了避免误解，一些人称这些类型为代数数据类型，因为构造器提供了一个代数，意味着“通过组合其他值来构建值的一组操作”。“多态数据类型”是指具有类型变量的数据类型，例如 `[(a, Bool)]`，与 `[Int]` 相对。有时人们会谈论“声明一个多态数据类型”或者说“`Maybe` 是一个多态数据类型”，其实他们只是想表达类型构造器有参数（因此可以用来形成多态类型）：严格来说，人们确实声明了一个多态数据类型，但并非任何普通的多态数据类型，而是一个应用于形式参数的类型构造器。

当然，所有按类型分类的一级值在某种意义上都是“数据”，在Haskell中，类型不会用于分类任何不是一级值的东西，因此在这个较宽松的意义上，每个“类型”都是“数据类型”。这个区分在其他语言中变得更有意义，在这些语言中，除了数据之外还有其他具有类型的事物（例如，Java中的方法）。

非正式的用法通常介于两者之间，且定义不太明确。人们常常试图区分函数或过程与它们操作的那种东西（“数据”）之间的某种区别。或者他们可能会认为数据是“通过它们的生成方式来理解的”（并暴露它们的表示，例如通过模式匹配），而不是“通过它们的使用方式来理解”。

“数据”这一最后的用法与抽象数据类型的概念有些不太契合，抽象数据类型是一种隐藏底层数据表示的类型。因此，隐藏表示的抽象数据类型与暴露表示的代数数据类型形成了鲜明的对比，这也是为什么“ADT”这个缩写被随意用来表示两者的情况相当不幸。

恐怕结论是模糊的。

## 1.17 你能将冒泡排序表述为一个么半群或半群吗？

我正在使用网络连接差的手机，但还是试试看。

`tl;dr` 冒泡排序是插入排序，是有序列表与合并的单集合的单集合“压缩”。

有序列表形成一个么半群。

```
newtype OL x = OL [x] instance Ord x => Monoid (OL x) where mempty
= OL [] mappend (OL xs) (OL ys) = OL (merge xs ys) where merge [] ys
= ys merge xs [] = xs merge xs@(x : xs') ys@(y : ys')
```

$\mid x \leq y = x : \text{合并 } xs' \quad ys \mid \text{否则} = y : \text{合并 } xs \quad ys$

插入排序由以下给出

```
isort :: Ord x => [x] -> OL x
isort = foldM ap (OL . pure)
```

因为插入操作正是将一个单元素列表与另一个列表合并。（归并排序是通过构建一个平衡树来实现的，然后进行相同的 foldMap。）

这和冒泡排序有什么关系？插入排序和冒泡排序采用完全相同的比较策略。如果你把它画成由比较并交换（compare-and-swap）盒子组成的排序网络，就可以看出来。在这里，数据向下流动，而盒子 [n] 的较低输入位于左侧：

```

| | | |
[1] | | [2]
| ] | [3] [4]
| [5] | [6]
| |
| | | |
```

如果按照上述编号给出的顺序进行比较，将图示切分成 7 个切片，就得到了插入排序：第一次插入不需要比较；第二次需要比较 1；第三次需要比较 2、3；最后一次需要比较 4、5、6。

但如果改为切成片...

```

| | | |
[1] | | [2]
| ] | [4] [3]
| [5] | [6]
| |
| | | |
```

……你在做冒泡排序：第一趟 1、2、3；第二趟 4、5；最后一趟 6。

## 1.18 这是一个在 Haskell 中正确实现的归并排序吗？

不，那不是 mergeSort。那是 insertionSort，从你怎么看它的角度来说，本质上与 bubbleSort 是同一种算法。每一步中，一个单元素列表都会与当前为止累积的有序列表合并，因此，实际上就是把该单元素列表中的元素插入进去。

正如其他评论者已经观察到的，要实现 mergeSort（尤其是其效率），必须反复将问题划分为大致相等的部分（而不是“一个元素”和“其余部分”）。“官方”解决方案提供了一种相当笨重的方法来做这一点。我非常喜欢

```
foldr (\x (ys, zs) -> (x : zs, ys)) ([],
```

抱歉，我需要更具体的文本来提供翻译。你可以再提供一下要

作为将列表分成两部分的一种方式，不是中间分割，而是将元素分成偶数和奇数位置。如果像我一样，你喜欢一开始就看到结构，那么你可以将有序列表转换为一个单一体。

```
import Data.Monoid import Data
.Foldable import Control.Newt
pe

newtype Merge x = Merge {merged :: [x]} instance Newt
ype (Merge x) [x] where pack = Merge unpack = merged
```

实例  $\text{Ord } x \Rightarrow$  单位元  $(\text{Merge } x)$  其中  $\text{mempty} = \text{合并 } []$   $\text{mappend } (\text{Merge } xs) (\text{Merge } ys) = \text{合并 } (\text{merge } xs \text{ } ys)$  其中  $-- \text{merge}$  是你定义的

现在你就只需通过

```
ala' 合并 foldMap (:[]) :: [x] -> [x]
```

获得  $\text{mergeSort}$  的分治结构的一种方法是把它做成一种数据结构：二叉树。

数据  $\text{Tree } x = \text{None} \mid \text{One } x \mid \text{Node } (\text{Tree } x) (\text{Tree } x)$  派生  $\text{Foldable}$

我在这里没有强制执行平衡不变量，但我可以。关键是，与之前相同的操作有另一种类型

```
ala' Merge foldMap (:[]) :: Tree x -> [x]
```

将从树状排列的元素收集的列表合并。为了获得这些排列，想一想“树的缺点是什么？”并确保保持平衡，通过我在上面的“分割”操作中使用的相同类型的扭曲。

```
扭曲 :: x -> 树 x -> 树 x -- 一种非常类似 cons 的类型 扭曲 x 无 = 一 x 扭曲 x (一 y)
= 节点 (一 x) (一 y) 扭曲 x (节点 l r) = 节点 (扭曲 x r) l
```

现在你通过构建一棵二叉树来实现归并排序，然后再将其合并。

```
mergeSort :: Ord x => [x] -> [x] mergeSort = 以 Merge 的方式 foldMap (:[]) . foldr
twistin None
```

当然，引入中间数据结构具有一定的好奇价值，但你可以很容易地将其去掉，得到类似于

```
mergeSort :: Ord x => [x] -> [x] mergeSort [] = [] mergeSort [x] = [x] mergeSort xs = mer
ge (mergeSort ys) (mergeSort zs) where (ys, zs) = foldr (\ x (ys, zs) -> (x : zs, ys)) ([], [])
xs
```

树已经成为程序的递归结构。

## 1.19 Haskell 类型系统的细微差别（歧义）

这是著名的展示. 阅读问题的一个变体。经典版本使用

```
阅读 :: 阅读一个 => 字符串 -> 一个显示
:: 显示一个 => 一个 -> 字符串
```

所以该组合看起来可能只是一个普通的 String 转换                      引导者

```
moo :: String -> String moo = show
w . read
```

除了程序中没有信息来确定中间的类型，因此无法决定读取什么并显示。

约束中的模糊类型变量 ‘b’：在…处使用 ‘read’ 引发的 ‘Read b’，在…处使用 ‘show’ 引发的 ‘Show b’。可能的修复方法：添加一个类型签名来固定这些类型变量。

请注意，ghci 做了一些疯狂的额外默认值处理，任意地解决歧义。

```
> (显示 . 读取) "()" "()"
```

您的 C 类是 Read 的变体，不同之处在于它解码一个 Int，而不是读取一个 String，但问题本质上是相同的。

类型系统爱好者会注意到，不受约束的类型变量并不是 *per se* 大问题。这里的问题是模糊性 *instance inference*。考虑

```
粪便 :: 字符串 -> a -> a 粪便 _ =
id
```

```
qoo :: (a -> a) -> 字符串 qoo _ = ""
```

```
roo :: 字符串 -> 字符串 roo = qoo
o . poo
```

在构造 roo 时，从未确定中间的类型必须是什么，roo 在该类型上也不是多态的。类型推断既不会求解该变量，也不会对其进行泛化！即便如此，

```
> roo "magoo"
```

这不是问题，因为该构造是 *parametric* 的未知类型。类型无法确定的事实意味着类型不能 *matter*。

但是未知的 *instances* 显然是重要的。Hindley-Milner 类型推理的完整性结果依赖于参数化，因此当我们添加重载时，该结果会丧失。让我们不要为此悲伤。

## 1.20 理解一个 Haskell 类型歧义的案例

这是一个经典问题。类型类的“特定用途”多态性使得类型推断不完全，而你刚好遇到了问题。我们来看看这些部分。

```
read :: Read x => 字符串 -> xflatten :: 嵌套列表 a -> [a]print :: Show y => y -> IO ()
```

而且我们还将有机器生成的实例用于  
 读取一个 => 读取 (NestedList a) 显示 a  
 => 显示 (NestedList a) 读取 a => 读取 [a]  
 显示 a=> 显示 [a]

现在让我们来解在尝试构建复合时得到的方程。

打印，扁平化，读取  $y = [a]$  嵌套列表  $a = x$

这意味着我们需要	
显示 [a]	阅读 (NestedList a)
因此	
显示一个	阅读一

并且我们在未确定 a 的情况下就使用了所有信息，因此也无法确定相关的 Read 和 Show 实例。

正如 J. Abrahamson 已经指出的那样，你需要做些什么来确定这个 `a`。有很多种方法可以做到这一点。我更倾向于使用类型注解，而不是编写那些唯一目的只是让类型更明显的奇怪项。我赞同给组合中的某个组件加上类型的提议，但我可能会选择 `(read :: String -> NestedList Int)`，因为正是这个操作引入了类型不明确的东西。

### 1.21 为什么 Haskell 使用 $\rightarrow$ 而不是 $=$ ?

这样写会很遗憾

$$(0, \_)=[]$$

因为那并不是真的。

在罗伯特·雷科德的传统中，我们仅在左侧等于右侧时才写出方程式。因此我们写道

重复  $x = (x, x)$

使  $\text{dup } x$  等于  $(x, x)$ , 或者

$$\text{dup} = \backslash x \rightarrow (x, x)$$

使 `dup` 等于将 `x` 映射到 `(x, x)` 的函数, 但并非

$$\backslash \mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$$

因为没有办法使  $x$  等于  $(x, x)$ 。

我们仅在允许“掉落”时稍微偏离传统，例如，

$$f(0) = 1$$
$$f_n = 2 * f_{(n-1)}$$

但只是指第二行中隐含了一个无声的“否则”。

## 1.22 如果类型参数的顺序错误，是否可以将一个类型实例化为一个类？

我可能有偏见，但我认为这是一个利用 `Control.Newtype` 的绝佳机会，它是一个只需执行 “`cabal install newtype`” 即可得到的小工具。

这是交易的内容。你想要翻转类型构造器，以便在不同的参数中获得函子性（例如）。定义一个新类型

```
newtype Flip f x y = Flip (f y x)
```

并将其添加到 `Newtype` 类中，如下

```
instance Newtype (Flip f x y) (f y x) where pack = Flip unpack (
  Flip z) = z
```

`Newtype` 类只是一个目录，将新类型映射到它们的原始等效类型，提供了方便的工具，例如，`op Flip` 是 `Flip` 的逆操作：你无需记住它被命名为什么。

对于这个问题，我们现在可以做这样的事情：

```
data Bif x y = BNil | BCons x y (Bif x y) deriving Show
```

这是一个带有两个参数的数据类型，恰好在这两个参数上都是函子性的。（也许我们应该把它做成 `Bifunctor` 类的一个实例，不过不管怎样……）我们可以把它变成一个 `Functor` 两次：一次针对最后一个参数……

实例 `Functor (Bif x)` 其中 `fmap f BNil = BNil` `fmap f (BCons x y b) = BCons x (f y) (fmap f b)`

... 并且第一次：

```
instance Functor (Flip Bif y) where fmap f (Flip BNil) = Flip BNil
fmap f (Flip (BCons x y b)) = Flip (BCons (f x) y (under Flip (fmap f) b))
```

其中，`under p f` 是对 `op p . f . p` 的一种简洁表述。我绝无虚言：让我们试试。

```
someBif :: Bif Int Char
someBif = BCons 1 'a' (BCons 2 'b' (BCons 3 'c' BNil))
```

然后我们得到

```
*翻转> fmap succ someBif BCons 1 'b' (BCons 2 'c' (BCons 3 'd' BNil))
*翻转> 下翻转 (fmap succ) someBif BCons 2 'a' (BCons 3 'b' (BCons 4 'c' BNil))
```

在这种情况下，同一件事确实可以通过许多方式被看作一个函子，所以我们确实有必要明确说明我们指的是哪一种方式。不过，只要你系统地处理，这种说明并不会带来太多额外的负担。

## 1.23 用于 Flip 数据类型的函子类型变量

翻译 `code` 就是现在，当你（使用 `ghc 8` 并且）打开一个标志 `o r` 两

```
Prelude> :set -XPolyKinds -XFlexibleInstances
```

让我们声明

```
Prelude> 新类型 Flip f a b = MkFlip (f b a)
```

然后询问

```
Prelude> :kind Flip Flip :: (k1 -> k -> *) -> k -> k1 ->
*
```

```
Prelude> :type MkFlip
```

```
MkFlip
  :: forall k k1 (b :: k) (f :: k -> k1 -> *) (a :: k1). f b a -> 翻转 f a b
```

`type` 构造函数 `Flip` 有两个隐式参数，分别是 `k` 和 `k1`，以及三个显式参数，分别是一个生成类型的二元函数，然后是它的两个参数，顺序颠倒。这个函数的参数是无约束类型（老一辈人可以随意称之为“种类”），但它肯定返回一个类型（在“\*中的东西”这一严格意义上，而不是“:: 右侧的任何陈旧废物”这一模糊无用的意义），因为它肯定作为类型在 `MkFlip` 的声明中使用。

`data` 构造子 `MkFlip` 接受 *five* 个隐式参数（正是 `Flip` 的那些参数），以及一个显式参数，即 `f b a` 中的某些数据。

正在发生的是高一层的 Hindley–Milner 类型推断。约束被收集起来（例如，`f b a` 必须属于 `*`，因为构造子参数必须属于 `f b a`），但除此之外会给出一个最一般的类型：`a` 和 `b` 可以是任意的，因此它们的类型被泛化为 `k1` 和 `k`。

让我们用常量类型构造器来玩同样的游戏：

```
Prelude> 新类型 K a b = MkK a
```

```
前奏> :kind K K :: * ->
k -> *
```

```
Prelude> :类型 MkK MkK :: 对于所有 k (b :: k) a. a ->
K a b
```

我们看到 `a :: *` 但 `b` 可以是任何旧的垃圾（而且，`k :: *`，因为现在 `* :: *`）。显然，`a` 实际上被用作某物的类型，但 `b` 根本没有被使用，因此没有约束。

我们可以然后声明

```
Pr 逃避> 实例函子（翻转 K b），其中 fmap f (MkFlip (MkK a)) = MkFlip (MkK (f a))
```

并询问

```
前奏> :info 翻转
... 实例 [safe] 对于所有 k (b :: k)。Functor (Flip K b)
```

这告诉我们，未使用的 `b` 仍然可以是任意的无关内容。因为我们有

$K :: * \rightarrow k \rightarrow *$  翻转  $:: (k1 \rightarrow k \rightarrow *) \rightarrow k \rightarrow k1 \rightarrow *$

我们可以统一  $k1 = *$  并得到

翻转  $K :: k \rightarrow * \rightarrow *$

因此

翻转  $K b :: * \rightarrow *$

对于任意的  $b$ 。因此，Functor 实例是合理的，而且确实可以实现的，其中函数作用于被打包的  $a$  元素，对应于  $\text{Flip } K b$  的参数，它成为  $K$  的 *first* 参数，因此也就是所存储元素的类型。

基于合一的类型推断在  $::$  的右侧依然（相当）健在。

## 1.24 为什么 `product []` 返回 1?

列表构成一个 *monoid* 结构，具有结合性的二元运算  $++$  以及中性元素 `[]`。也就是说，我们有

$[] ++ xs = xs = xs ++ []$        $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$

与此同时，数具有许多么半群结构，但这里相关的是运算为  $*$ 、单位元为 1 的那个。

$1 * x = x = x * 1$        $(x * y) * z = x * (y * z)$

乘积函数不仅仅是从数字列表到数字的映射：它是一个 *monoid homomorphism*，将列表么半群结构反映到数值么半群中。至关重要的是，

乘积  $(xs ++ ys) = \text{乘积 } xs * \text{乘积 } ys$

和

产品  $[] = 1$

事实上，为了得到前者，我们几乎是被迫接受后者。

## 1.25 两个 Maybe 的最小值

可以使用 `Control.Applicative` 中的运算符来满足该规范。

$\text{myMin} :: \text{Ord } x \Rightarrow \text{Maybe } x \rightarrow \text{Maybe } x \rightarrow \text{Maybe } x$   
 $\text{myMin } a \text{ } b = \text{mi}$   
 $n <\$> a <*> b <|> a <|> b$

其中 `Maybe` 的 `<|>` 实现了“偏好”

什么都没有 `<|> b = b a`

`<|> _ = a`

问题是

最小值 `<$>` 仅 `a <*>` 仅 `b = 仅 (min a b)`

但是

## 1.26. IS THE EQUIVALENT OF HASKELL'S FOLDABLE AND TRAVERSABLE SIMPLY A SEQUENCE IN CLOJURE?

最小值 `<*>` 只是一个 `<*>` 什么也没有 = 什么也没有

这已经导致了对该问题的一些错误答案。使用 `<|>` 允许你在可用时优先选择计算得到的最小值，但当只有一个为 `Just` 时，可以用任一单独值进行回退。

但你应该问问是否适合以这种方式使用 `Maybe`。除了它的 `Monoid` 实例外，`Maybe` 被设置为建模容易失败的计算。你这里所拥有的是对现有 `Ord` 的扩展，带有一个“top”元素。

数据 `Topped x = Val x | Top` 派生 (`Show, Eq, Ord`)

你会发现 `Topped x` 的最小值正是你需要的。将类型视为不仅仅是数据的表现形式，而是赋予数据结构的工具，这种思维方式是有益的。通常，`Nothing` 表示某种失败，因此为你的目的使用不同的类型可能更好。

## 1.26 Haskell 的 Foldable 和 Traversable 在 Clojure 中的等价物是否只是一个序列？

不是。虽然任何表示有限元素序列的 `Functor` 都是 `Traversable`（因此也是 `Foldable`），但还有许多其他结构也是 `Traversable`，却并非类序列的，因为它们没有明显的拼接概念。会有一种方法可以获取其中所包含元素的序列，但该结构可能不止由这个序列构成。

`Traversable f` 的意思实际上是，类型为 `f x` 的结构包含有限个类型为 `x` 的元素，并且有某种方式可以遍历该结构，使每个 `x` 元素恰好访问一次。所以像“语法中的项，被看作包含变量”这样的东西是可以是 `Traversable` 的。

数据术语 `x`  
= 变量 `x` | 值 整数 | 加 (项 `x`) (  
项 `x`)

为 `Term` 定义 `Traversable` 实例

遍历 `f` (变量 `x`) = 纯 变量 `<*>` `f x` 遍历 `f` (值 `i`) = 纯 (值 `i`) 遍历 `f` (加 `s t`) = 纯 加 `<*>` 遍历 `f s <*>` 遍历 `f t`

你始终可以使用 `traverse` 对所有元素进行操作。我们通过将 `pure = id` 和 `<*>` 视为普通应用来获得 `fmap`。

```
instance Functor Term where fmap = fm
apDefault
```

在哪里

```
fmap :: (x -> y) -> 项 x -> 项 y
```

实现了 *simultaneous renaming*。同时  
, `Foldable` 实例

```
instance Foldable Term where foldMap =
foldMapDefault
```

将 `pure` 映射为某个么半群的单位元，并将 `<*>` 映射为组合运算，因此我们得到类似 `reduce` 的操作。例如，

```
foldMap (:[]) :: Term x -> [x]
```

给出了出现在项中的变量列表。也就是说，我们可以始终从可遍历数据中获取元素序列，但数据可能具有不同于这些元素的结构。项具有不同于变量的结构（它们的值和值的添加），而且对于语法树来说，“cons”意味着什么并不十分清楚。

所以，尽管比起序列，更多的结构是可遍历的，但 Traversable 接口提供的序列式操作却较少。Traversable 的重点是将 *map* 类似和 *reduce* 类似的操作进行泛化，而不是捕捉 *list* 特性。

## 1.27 如何在不多次遍历字符串的情况下跟踪字符串的多个属性？

我将从定义传统的“示性函数”开始

```
指示 :: 数字 a => 布尔 -> a 指示 b = 如果 b
则 1 否则 0
```

以便

```
指示 . isVowel :: Char -> 整数
```

接下来，我会从 Control.Arrow 获取两件关键设备。

```
(&&&) :: (x -> y) -> (x -> z) -> x -> (y, z) (***) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)
```

所以（记住有些字符既不是元音也不是辅音）

```
(indicate . isVowel) &&& (indicate . isConsonant) :: 字符 -> (整数, 整数)
```

然后我会从 Data.Monoid 中获取 Sum。

```
(Sum . indicate . isVowel) &&& (Sum . indicate . isConsonant) :: Char -> (Sum Integer, Sum Integer)
getSum *** getSum :: (Sum Integer, Sum Integer) -> (Integer, Integer)
```

现在我部署 y foldMap，因为我们正在做某种单侧的“压碎”。

```
(getSum *** getSum) . foldMap ((Sum . indicate . isVowel) &&& (Sum . indicate . isConsonant)) :: String -> (Integer, Integer)
```

然后我记得我写了一些代码，这些代码被转化成了 Control.Newtype，后来我发现以下内容缺失了，但应该存在。

```
实例 (Newtype n o, Newtype n' o') => Newtype (n, n') (o, o') 其中 pack = pack *** pack unpack
k = unpack *** unpack
```

而现在我只需要写下

```
ala' (Sum *** Sum) foldMap ((indicate . isVowel) &&& (indicate . isConsonant)) :: String -> (Integer, Integer)
```

关键小工具是

```
ala' :: (Newtype n o, Newtype n' o') => (o -> n) -> ((a -> n) -> b -> n') -> (a -> o) -> b -> o' -- ^packer ^higher-order operator ^action-on-elements
```

其中，打包器的职责是选择具有正确行为实例的 `newtype`，并同时确定解包器。它正是为了支持在更具体的类型上进行本地化工作而设计的，该类型能够指示预期的结构。

## 1.28 检查一棵二叉树是否为二叉搜索树

这里有一种无需将树扁平化的方法。根据定义，  
这里，

数据 二叉树 `a = 空 | 节点 (二叉树 a) a (二叉树 a)` 派生 显示

可以看出，从左到右遍历树，忽略节点和括号，会得到一个交替的 `Null` 和 `as` 的序列。也就是说，在每两个值之间，都会有一个 `Null`。

我的计划是检查每个子树是否满足适当的 *requirements*：我们可以 *refine* 每个节点的要求，记住我们之间的值，然后在每个 `Null` 处 *test* 它们。由于在每对顺序值之间都有一个 `Null`，我们将测试所有顺序（从左到右）对是否是非递减的。

什么是约束？它是对树中取值的 *loose* 下界和上界。为了表达约束（包括位于最左端和最右端的约束），我们可以通过引入 `Bottom` 和 `Top` 元素来扩展任意排序，如下所示：

数据 `TopBot a = Bot | Val a | Top` 派生 (`Show, Eq, Ord`)

现在让我们检查一棵给定的树是否满足既有序又在给定边界之间的要求。

```
ordBetween :: Ord a => TopBot a -> TopBot a -> BinaryTree a -> Bool -- 收紧所要求的边界，适用于任意 Node 的左侧和右侧
ordBetween lo hi (Node l x r) = ordBetween lo (Val x) l && ordBetween (Val x) hi r -- 当到达 Null 时，检查所要求的边界是否有序
ordBetween lo hi Null = lo <= hi
```

一个二叉搜索树是一个有序的树，位于 `Bot` 和 `Top` 之间。

```
isBSTree :: Ord a => 二叉树 a -> 布尔
isBSTree = ordBetween Bot Top
```

在每个子树中计算 *actual* 的极值并将其向外冒泡，会提供超过所需的信息，而且在左或右子树为空的情况下相当繁琐。维护并检查 *requirements*，将它们向内推进，则要统一得多。

## 1.29 在二叉树中查找值为 x 的叶子节点

您的语言，以及您认为程序应该 *do* 的内容，暗示我您需要帮助，摆脱命令性思维的陷阱。让我尝试提供一些帮助，基于思考应该做什么 *are*，而不是做什么 *do*。

对于 `findpath (Leaf y) x`，你正朝着正确的方向前进。你只需要给它一个小写的 `i`，并思考通向一个 `Leaf` 的正确 `Path` 应该是什么。

现在，让我们考虑另一种可能性。你知道的不仅仅是它是某个 `t`。你知道你实际上是在努力弄明白什么

查找路径 (节点 `l r`) `x`

是（这就是 `=`，确实如此），因为那是 `B` 树的另一种可能性。可以通过提问“这个 `B` 树是（叶子 `y`）还是（节点 `l r`）？”来分解问题，这是程序设计中的一个概念性步骤。现在，为了弄清楚上面左边的等式等于什么，你有权获得一些递归计算的信息，即

查找路径 `l x`

和

查找路径 `r x`

如果你知道 `l` 和 `r` 的路径信息，你能说出整个节点 `l r` 的路径是什么吗？让我通过将这个问题用 `Haskell` 重新表述：

```
findpath :: Eq a => BTree a -> a -> Path
findpath (Leaf y) x = if y==x then Just [] else Nothing
findpath (Node l r) x = nodepath (findpath l x) (findpath r x)
其中 nodepath :: Path -> Path -> Path
nodepath l r = if l==r then Just l else Nothing
```

我通过引入一个 *helper function* 节点路径来表达我的问题，该节点路径接受递归计算的信息作为参数。现在，你可以尝试通过模式匹配分别对左右子树的这两条路径进行实现。如果你知道它们是 `(Just p)` 还是 `Nothing`，那么你应该能够确定整个节点的路径是什么。

第一课，有用的思想具有如下形式：“如果这个像某某那样，那么那个就必然是某某那样。”。存在，而非行动。

第二课，对一个数据类型进行编程的基本方法是：按构造子情况进行划分（`Leaf` 对 `Node`，`Just` 对 `Nothing`）；通过递归调用从任何子结构中收集有用的信息；说明整个结构的取值应当是什么。

如果你遵循我的建议并弄清楚 `nodepath` 应该是什么，你可能会发现它足够简单，不值得单独作为命名定义。在这种情况下，只需用其含义替换 `nodepath` 调用，并去掉 `where` 子句。但从引入 `nodepath` 开始是好的，因为它表达了解决问题的有用概念步骤。

## 1.30 从列表中取值直到遇到重复项

你指出 `takeWhile` 之所以不起作用，是因为你没有各个元素的上下文信息，这暗示了如下策略：获取这些信息。

我的这个答案涉及到装饰带上下文操作，我称之为 `picks`（因为它向你展示了如何选择一个元素进行聚焦）。它是我们理应为每个容器对象免费提供的通用装饰带上下文操作。对于列表，它是

```
picks :: [x] -> [(x, ([x], [x]))] -- [(x-before, ([x-before], [x-after]))]
picks [] = []
picks (x : xs) = (x, ([], xs)) : (y : ys, zs) | (y, (ys, zs)) <- pick
```

ks xs]

它对于无限列表非常有效，顺便说一下。现在试试这个

```
takeUntilDuplicate :: Eq x => [x] -> [x]
takeUntilDuplicate = map fst . takeWhile (\ (x, (ys, _)) -> not (elem x ys)) . picks
```

(奇怪的是，如果不给出上述类型签名，上面的那条单行代码会因为 `Eq` 的歧义而被拒绝，这让我感到不安。我很想在这里就此提个问题。Oh, it's the monomorphism restriction. How annoying.)

备注。使用 `snoc`-列表（从右侧增长的列表）来表示“之前的元素”组件，这样可以更好地保持共享性并保留从左到右的视觉顺序，这样做是非常有意义的（而且我通常会这么做）。

## 1.31 RankNTypes 与 PolyKinds（量词交替问题）

让我们保持血腥。我们必须量化一切并给出量化的领域。值有类型；类型级别的东西有种类；种类存在于 `BOX` 中。

```
f1 :: 对于所有 (k :: BOX)。 (对于所有 (a :: k) (m :: k -> *) . m a -> Int) -> Int
```

```
f2 :: (forall (k :: BOX) (a :: k) (m :: k -> *) . m a -> Int) -> Int
```

现在，在这两种示例类型中，`k` 都没有被显式量化，因此 `ghc` 会根据 `k` 是否以及在何处被提及来决定把那个 `forall (k :: BOX)` 放在哪里。我并不完全确定我是否理解，或者是否愿意为如上所述的这一策略辩护。

Ørjan 给出了一个很好的实践差异示例。我们也要对这一点直言不讳。我将写  $\bigwedge (a :: k). t$  来明确表示对应于 `forall` 的抽象，并且用 `f @ type` 来表示对应的应用。游戏的规则是，我们可以选择 `@-ed` 参数，但我们必须准备好接受魔鬼可能选择的任何  $\bigwedge$ -ed 参数。

我们有

```
x :: 对于 (a :: *) (m :: * -> *) . m a -> 整型
```

并可能因此发现 `f1 x` 其实是

```
f1 @ * (\ (a :: *) (m :: * -> *) . x @ a @ m)
```

然而，如果我们尝试对 `f2 x` 进行同样的处理，我们会看到

```
f2 (\ (k :: BOX) (a :: k) (m :: k -> *) . x @ ?m0 @ ?a0) ?m0 :: * ?a0 :: * -> * 其中
m a = m0 a0
```

Haskell 类型系统将类型应用视为纯粹的语法问题，因此解决该方程的唯一方法是通过识别函数和识别参数。

```
(?m0 :: * -> *) = (m :: k -> *)
(?a0 :: *)      = (a :: k)
```

但那些方程甚至都不是良好定类的，因为 `k` 并不能被自由选择：它被  $\bigwedge$ -ed，而不是 `@-ed`。

通常，为了掌握这些超多态类型，最好是写出所有的量词，然后弄清楚它是如何转化为你与魔鬼之间的博弈的。谁选择什么，以及选择的顺序。将一个 `forall` 移入参数类型会改变它的选择者，并且通常能决定胜负。

## 1.32 如何使用视图模式语法来编写这个 case 表达式?

它们并不等价。case 版本只有一个 readMaybe，而视图模式版本有两个。对于每一个 readMaybe，编译器都必须推断尝试读取时所针对的类型。当代码写成这样时

```
parse xs x = case readMaybe x of Just x -> Right x
ht (x : xs) Nothing -> Left "语法错误"
```

GHC 调试器注意到，在你的 Just x 情况下，x 最终被 cons 到 xs 中，因此必须采用 xs 中元素的类型。这是很好的工作。但当你写出

```
parse xs (readMaybe -> Just x) = Right (x : xs)
parse xs (readMaybe -> Nothing) = Left "语法错误"
```

你创建了两个独立的“寻找目标”类型问题，每个问题对应一次使用 readMaybe。第一个问题的解决方式与 case 情况中的方法相同，但对于第二个问题，要分别处理。

部分 `e xs (readMaybe -> Nothing) = Left "语法错误"`                      错误

没有任何线索 *what* 是你未能阅读的，也没有理由相信它与上面那一行的内容是一样的。

一般来说，除非只有一个感兴趣的结果，否则使用视图模式是不恰当的。如果你想进行一次中间计算 *once*，但要将结果分析为多种情况，那么它们就是错误的语法。基于这个原因，我乐于公开表明我认为它们是一个设计缺陷。

## 1.33 递归类型族

类型推断默认是一种猜测游戏。Haskell 的表面语法使得在明确指定应实例化哪个类型时显得相当笨拙，即使你知道自己想要什么。这是 Damas-Milner 完备性的遗产，在那个时代，足够有趣的概念要求显式类型时，简直是被禁止的。

让我们想象我们被允许在模式和表达式中将类型应用显式化，使用一种 Agda 风格的  $f\{a = x\}$  记法，从而有选择地访问  $f$  的类型签名中与  $a$  对应的类型参数。你的

```
idT = StateT $ \ s -> idT
```

应该意味着

```
idT {a = a}{m = m} = StateT $ \ s -> idT {a = a}{m = m}
```

使得左边的类型是  $C\ a\ a\ (StateT\ s\ m)\ r$ ，而右边的类型是  $StateT\ s\ (C\ a\ a\ m)\ r$ ，根据类型族的定义它们是相等的，欢乐便在世界上传播开来。但这并不是你所写内容的含义。用于调用多态事物的“变量规则”要求：每一个 forall 都要用一个新的存在类型变量进行实例化，然后再通过统一化来求解。所以你的代码所表达的意思是

```
idT {a = a}{m = m} = StateT $ \ s -> idT {a = a'}{m = m'}
-- for a suitably chosen a', m'
```

在计算类型族之后，可用的约束是

```
C a a m C a' a' m'
```

但是那并不会被简化，也不应该被简化，因为没有理由假设  $C$  是单射。令人抓狂的是，机器比你更在意找到一个最一般解的可能性。你心中已经已经有了一个合适的解，但问题在于，当默认假设是 *guesswork* 时，要实现 *communication*。

有许多策略可以帮助你摆脱困境。其中之一是改用数据族。优点：单射没有问题。缺点：结构问题。（警告，以下是未经测试的推测。）

```
类 MonadPipe m 其中 数据 C a b (m :: * -> *) r idT :: C a a m r (<-> :: C b c m r -> C a b m r -> C a c m r
```

```
实例 (MonadPipe m) => MonadPipe (StateT s m) 其中 数据 C a b (StateT s m) r = StateTPipe (StateT s (C a b m) r) idT = StateTPipe . StateT $ \ s -> idT StateTPipe (StateT f) <-< StateTPipe (StateT g) = StateTPipe . StateT $ \ s -> f s <-< g s
```

另一个缺点是，得到的数据族并不会自动成为单子，而且要对其解包，或以统一的方式将其单子化，也并不那么容易。

我在考虑尝试一种模式，在这种模式中，你保持你的字体家族，但为其定义一个新的类型包装器。

```
新类型 WrapC a b m r = WrapC {unwrapC :: C a b m r}
```

然后在操作类型中使用 `WrapC`，以保持类型检查器走在正确的轨道上。我不知道这是否是一个好策略，但我打算找出答案，某天会明白的。

一种更直接的策略是使用代理、幻影类型和作用域类型变量（尽管这个例子不需要它们）。（再次提醒，属于猜测。）

```
数据代理 (a :: *) = 代理数据 ProxyF (a :: * -> *) = 代理F
```

```
类 MonadPipe m 其中 数据 C a b (m :: * -> *) r idT :: (Proxy a, ProxyF m) -> C a a m r ...
```

```
实例 (MonadPipe m) => MonadPipe (StateT s m) 其中数据 C a b (StateT s m) r = StateTPipe (StateT s (C a b m) r) idT pp = StateTPipe . StateT $ \ s -> idT pp
```

那不过是一种把类型应用显式化的蹩脚做法。注意，有些人使用 `a` 本身而不是 `Proxy a`，并把 `undefined` 作为参数传入，从而未能在类型中将其标记为代理，而是依赖于不会意外对其求值。PolyKinds 的最新进展或许至少意味着我们可以只拥有一种种类多态的幻影代理类型。关键的是，`Proxy` 类型构造子是单射的，因此我的代码确实是在说“这里和那里是相同的参数”。

但有时我希望能够在源代码中降到 `System FC` 级别，或者以其他方式清晰地表达对类型推断的手动覆盖。这样的事情在

依赖类型社区，在这里没有人认为机器可以在没有一点提示的情况下解决所有问题，或者隐藏的参数没有值得检查的信息。在使用位置中，函数的隐藏参数通常可以被省略，但在定义中需要显式地指出。当前Haskell的情况基于一种文化假设，即“类型推断足够了”，这种假设已经偏离轨道一代人，但仍然以某种方式持续存在。

## 1.34 确定一个值是否是Haskell中的函数

参数化性说不。唯一具有该类型的函数

`a -> 布尔`

是常量函数。然而，借助一点特设多态性，再多一点胆量，你可以这样做：

```
``haskell{-# LANGUAGE 重叠实例, 灵活实例 #-}``
```

```
类 Sick x 其中 isFunc :: x -> Bool
```

```
实例 Sick (a -> b) 其中 isFunc _ = 真
```

```
实例 Sick x, 其中 isFunc _ = False
```

然后看起来你有

```
*Sick> isFunc 3 False *
Sick> isFunc id True
```

但这确实看起来是个古怪的做法。得到的 `Bool` 对你有什么用？

## 1.35 自动函子实例（不）

在 `ghci` 中，如果你念出咒语

```
前奏> :set -XDeriveFunctor
```

那么编译器就会变得和你所期望的一样聪明，尽管可能没有那么热情。你需要调用该功能，因此，

```
前奏> 数据 Foo a = Foo a 派生 (显示, 函子)
```

翻译文本：(该显示仅用于打印输出，下面) 然后你将能够做一些事情，例如

```
前奏> fmap (++"bar") (Foo "foo") Foo "foobar"
```

在模块中，您通过添加指令实现相同的功能

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

### 1.36. 我如何应用在 Haskell 中有效的第一个偏函数? 39

在模块声明之前。这对至少更直接的函子实例是有效的，但你可以使它产生错误的负面结果。

前奏> 数据 Boo 一个 = Boo (要么是一个布尔值) 衍生 自由Functor

<interactive>:9:43: 无法创建 ‘Functor Boo’ 的派生实例：构造函数 ‘Boo’ 必须仅将类型变量作为数据类型的最后一个参数。在 ‘Boo’ 的数据声明中

同时

数据一个 Goo 一个 = Goo (Either Bool a) 派生 Functor

没问题，而且机械显然已经被破解，以便与配对一起使用，

数据 Woo a = Woo (a, 布尔) 派生 函子

是允许的。所以它并不像它本可以那么聪明，但总比眼睛里被戳一刀好。

### 1.36 如何应用在 Haskell 中有效的第一个偏函数?

在 Data.Monoid 中，Maybe 的一个新类型副本，称为 First，具有“取第一个 Just”的行为。

如果你在寻找一种类型的函数{v\*}

[a -> 第一个 b] -> a -> 第一个 b

根据你描述的行为，它将简单地是

折叠

来自 Data.Foldable，因为 a -> 的单元行为执行了所需的逐点提升：a -> 的 Monoid First b 正是选择第一个应用结果有效的地方。可悲的是（为此我哭泣过很多次），要获得 Maybe 而不是 First 需要更多的工作。

请注意，逐点提升，将->从[]中拉出来，正是sequenceA的工作，所以

(asum .) . sequenceA

会完成这项工作。

从类型中获取所需的单子结构是很好的提示：在这种情况下，访问 Alternative 行为时，asum 就足够了。

### 1.37 ‘压缩’ 一个包含嵌套列表的普通列表

这样可以吗?

翻转 . (evalState .) . traverse . traverse . const . state \$ head &&& tail

编辑：让我详细说明一下这个构造……

它的本质中心是遍历。遍历。如果你用足够差的眼镜盯着这个问题看，你会看到它是“对一个容器的容器的元素做些什么”。对于这种事情，遍历（来自Data.Traversable）是一个非常有用的工具（好吧，我有偏见）。

```
traverse :: (Traversable f, Applicative a) => (s -> a t) -> f s -> a (f t)
```

或者，如果我改用更长但更具描述性的类型变量

```
traverse :: (Traversable containerOf, Applicative doingSomethingToGet) => (s -> doingSomethingToGet t)
-> containerOf s -> doingSomethingToGet (containerOf t)
```

关键在于，traverse 会保留它所操作的容器的结构，不论那是什么。如果你将 traverse 视为一个高阶函数，就会看到它返回的是一个作用于容器的算子，其类型与它所要求的、作用于元素的算子的类型相匹配。也就是说，(traverse . traverse) 是有意义的，并且会为你提供对 *two* 层容器进行结构保持的操作。

遍历 . 遍历 ::

```
(可遍历 g, 可遍历 f, 应用函子 a) => (s -> a t) -> g (f s) -> a (g (f t))
```

所以我们就有了用于结构保持的“做某事”操作的关键工具，适用于列表的列表。长度和 splitAt 方法在列表上工作得很好（列表的结构由其长度给出），但使得该方法可行的列表的基本特性已经被 Traversable 类基本封装起来。

现在我们需要弄清楚如何“做点什么”。我们想用从一个供给流中依次取出的新事物来替换旧的元素。如果允许通过更新供给产生副作用，我们就可以在每个元素处说明该做什么：“返回供给的头部，并将供给更新为其尾部”。State s 单子（位于 Control.Monad.State 中，并且是 Applicative 的一个实例，来自 Control.Applicative）让我们能够捕捉这一想法。类型 State s a 表示这样的计算：在改变类型为 s 的状态的同时，产出一个类型为 a 的值。典型的这类计算由这个小工具构造。

```
状态 :: (s -> (a, s)) -> State s a
```

也就是说，给定一个初始状态，只需计算该值以及新的状态。在我们的例子中，s 是一个流，head 得到值，tail 得到新的状态。&&& 运算符（来自 Control.Arrow）是一种很好的方式，用于在同一份数据上把两个函数粘合起来，从而得到一个生成二元组的函数。因此

```
head &&& tail :: [x] -> (x, [x])
```

这使得

```
状态 $ 头 &&& 尾 :: 状态 [x] x
```

因此

```
const . state $ head &&& tail :: u -> State [x] x
```

解释了如何“处理”旧容器的每个元素，即忽略它并从供应流的开头获取一个新元素。

将其输入到 (traverse . traverse) 中，我们得到一个大范围的变异遍历，类型为

```
f (g u) -> State [x] (f (g x))
```

其中  $f$  和  $g$  是任意的 `Traversable` 结构（例如列表）。

现在，为了提取我们想要的函数，采用初始供应流，我们需要将状态变更计算解包为从初始状态到最终值的函数。这就是它的作用：

```
evalState :: State s a -> s -> a
```

所以我们最终得到的是某种在

```
f (g u) -> [x] -> f (g x)
```

如果要匹配原始规格，最好把它翻转过来。

`tl;dr State [x]` 单子是一种现成的工具，用于描述读取并更新输入流的计算。`Traversable` 类型类捕捉了对容器进行结构保持操作的一种现成概念。其余都是管道活（和/或代码高尔夫）。

## 1.38 成簇堆积

我看到我们正在对某种数据结构进行累积。我想到了 `foldMap`。我问：“哪种单元？”它是某种累积列表。像这样

```
新类型 Bunch x = Bunch {bunch :: [x]} 实例 Semigroup x => Monoid (Bunch x) 其中 me
mpty = Bunch [] mappend (Bunch xss) (Bunch yss) = Bunch (glom xss yss) 其中 glom [] ys
s = yss glom xss [] = xss glom (xs : xss) (ys : yss) = (xs <> ys) : glom xss yss
```

我们的底层元素有一些关联操作符 `<>`，因此我们可以像 `zipWith` 一样，按点将该操作符应用于一对列表，只是当其中一个列表用尽时，*we don't truncate*，我们直接取另一个列表即可。请注意，`Bunch` 是我为回答这个问题而引入的一个名称，但它并不是一个不寻常的需求。我相信我之前用过它，以后也会再用。

如果我们能够翻译

```
0 -> 束 [0] -- 在位置0处的单个0
0 1 -> 束 [0,[1]] -- 在位置1处的单个1
1 2 -> 束 [0,[1],[2]] -- 在位置2处的单个2
2 3 -> 束 [0,[1],[2],[3]] -- 在位置3处的单个3
```

...

并在整个输入上进行 `foldMap`，这样我们就在每个位置得到每一种元素的正确数量。为了得到合理的输出，并不需要对输入中的数字设定上界，只要你愿意将 `[]` 解释为“其余皆为沉默”。否则，就像普罗克鲁斯忒斯之床一样，你可以通过填充或裁剪来达到所需的长度。

请注意，当 `mappend` 的第一个参数来自我们的翻译时，我们会进行一系列 `([]++)` 操作，也就是 `ids`，然后进行一次 `([i]++)` 操作，也就是 `(i:)`，所以如果 `foldMap` 是右嵌套的（对于列表来说确实是这样），那么我们将始终在列表的左端进行廉价操作。

现在，由于这个问题是围绕列表展开的，我们可能只在它有用的时候才引入 `Bunch` 结构。这正是 `Control.Newtype` 的用途。我们只需要把 `Bunch` 告诉它即可。

```
instance Newtype (Bunch x) [x] where
    pack = Bunch
    unpack = bunch
```

然后就是

```
groupInts :: [Int] -> [[Int]]
groupInts = ala' Bunch foldMap (basis !!) where basis = ala' Bunch foldMap
id [iterate ([:] []) , [[i]] | i <- [0..]]
```

什么？好吧，在不详细讲解 `ala'` 一般意义的情况下，它在这里的影响如下：

```
ala' Bunch foldMap f = bunch . foldMap (Bunch . f)
```

意味着，尽管 `f` 是一个作用于列表的函数，我们却像 `f` 是一个作用于束的函数一样进行累积：`ala'` 的作用是插入正确的打包和解包操作，使得这一切得以实现。

我们需要 `(basis !!) :: Int -> [[Int]]` 作为我们的翻译。因此 `basis :: [[Int]]` 是我们翻译的图像列表，根据需要计算，每个最多计算一次（即翻译，*memoized*）。

对于这个基础，观察到我们需要这两个无限列表  $\{v^*\}$

Sorry, it seems like the source text is empty. Could you please provide the text you'd like to have translated?

, [[]]	, [[1]]
, [[], []]	翻译文本: , [[2]]
, [[], [], []]	, [[3]]
...	...

按束合并。由于两个列表具有相同的长度（无穷），我也可以写成

```
basis = zipWith (++) (iterate ([:] []) [[i]] | i <- [0..])
```

但我认为值得注意的是，这同样也是 `Bunch` 结构的一个例子。

当然，当像 `accumArray` 这样的函数能准确地为你提供所需的累积，并巧妙地包装了一堆脏乱的幕后变更时，当然是非常不错的。但累积的一般思路是思考“什么是单元？”，以及“我该如何处理每个元素？”这正是 `foldMap` 问你要做的事情。

## 1.39 幻影类型上的函子

在函数式编程中，一个关键技能是在特定情境下掌握正确的“什么都不做”的方式。经典错误的产生，例如，来自于将 `[]` 作为递归搜索的基准情况表示“没有解”，而实际上需要的是 `[[]]`，表示“一个琐碎的解”。因此，幽灵类型的函子实例，即常量函子，作为更大模式的看似琐碎的基准情况，也非常有用。

我可以如下介绍用于构建类似容器结构的通用工具集：

```
newtype K a x = K a deriving Functor -- K 代表常量 {- fmap _ (K a) = K a -}
```

```
newtype I x = I x 派生 Functor -- I 表示恒等
{- fmap k (I x) = I (k x) -}
```

```
newtype P f g x = P (f x, g x) deriving Functor -- P 用于积 {- 将给出 (Functor f, Functor g) =>
Functor (P f g), 使得 fmap k (P (fx, gx)) = P (fmap k fx, fmap k gx) -}
```

新的类型 `S f g x = S (Either (f x) (g x))` -- `S` 用于和类型实例 `(Functor f, Functor g) => Functor (S f g)` 其中 `fmap k (S (Left fx)) = S (Left (fmap k fx))` `fmap k (S (Right gx)) = S (Right (fmap k gx))`

现在，任何递归数据结构都可以表示为一个顶层节点，该节点充当子结构的容器。

数据 数据 f = 节点 (f (数据 f))

例如，如果 我想用数字在叶子上构建二叉树， 我可以写

类型 Tree = S (K Int) (P I I)

用于表明一棵树的节点结构要么是带有一个 Int 和 *no subtrees* 的叶子，要么是包含一对子树的分叉。我需要 K 指出递归子结构的 *absence*。于是树的类型就是 Data Tree。

这些事情的通常递归方案是

折叠 :: 函子 f => (f t -> t) -> 数据 f -> t 折叠 k (节点 fd) = k (fm  
ap (折叠 k) fd)

我们不需要为树去做任何实例化工作，因为 Tree 已经是一个 Functor，这是因为它是由具有函子性的组件构建而成的。对 K Int 的平凡 fmap 等同于说明：当你到达一个叶子时，递归就会停止。

当然，这些“编码”的数据类型会让你在通过模式匹配编写普通程序时更难看清自己在做什么。这时 PatternSynonyms 扩展就会来拯救你。你可以说

模式 Leaf i = 节点 (S (Left (K i))) 模式 Fork l r = 节点 (S (Right (P (I l, I r  
))))

以恢复常规界面。我建议去掉外层 Node，以便契合 fold 为你剥离 Node 的方式。

模式 叶 i = S (左 (K i)) 模式 叉 l r = S (右 (P (I l, I r)))

添加 :: 数据树 -> 整数 添加 = 折叠 \$ \ t -  
> 情况 t 的 Leaf i -> i Fork x y -> x + y

我几乎只是触及了皮毛：只要你能仅为 K、I、P 和 S 开发这些功能，就可以将这类通用功能推广到许多数据类型。K 的情况总是微不足道，但它们必须存在。

类似的考量也适用于 Void 数据类型（在 Data.Void 中）。我们究竟为什么要费心引入一种没有任何值得一提的元素的数据类型？为了对更大方案中不可能的情况进行建模。

## 1.40 在 Haskell 中创建一个解释器（带存储）

回答你的问题有点困难，因为你实际上并没有提出一个问题。让我挑选出你说的一些内容，以便给你一些线索。

我不确定在这个问题中是否需要使用 evalE。我在之前的一个问题中已经写过它。eval  
E 的类型签名是 evalE :: Expression -> Store -> (Value, Store)

evalS (Expr e) s = .....不确定这里该做什么。

执行一个由表达式组成的语句是什么意思？如果它与评估表达式有关，那么你有一个表达式评估器可用，可能有助于“在这里该做什么”。

接下来，比较一下你被给出的“while”的代码（顺便一提，其中包含了一个对表达式进行合理处理的很好示例）。...

```
evalS w@(While e s1) s = case (evalE e s) of ' (BoolVal True,s') -> let s' ' = evalS
s1 s' in evalS w s' ' (BoolVal False,s') -> s' _ -> error "条件必须是 BoolVal"
```

... 并将其与你的“if”代码进行比较

```
evalS (If e s1 s2) s = 执行 x <- evalE e 对 x
进行匹配 BoolVal True -> evalS s1 BoolVal
False -> evalS s2
```

你的代码采用了一种相当不同的风格——“单子式 (monadic)” 风格。你是从哪里得到这种风格的？如果求值器的类型大致是下面这样的，那就说得通了

```
evalE :: 表达式 -> 状态 存储 值 evalS :: 语句 -> 状态 存
储 ()
```

单子风格是一种非常好的方式，可以在不必过多谈论的情况下，将可变存储在求值过程中进行传递。例如，你的 `x <- evalE e` 的意思是“令 `x` 为对 `e` 求值的结果（悄然接收初始存储并将得到的存储继续传递下去）”。这是一种很好的工作方式，我预计你会在适当的时候进一步探索。

但这些并不是你被给予的类型，而且单子风格并不合适。你有

```
evalE :: 表达式 -> 存储 -> (值, 存储) evalS :: 语句 -> 存储 -> 存
储
```

而示例代码显式地传递了 `store`。再看一遍

```
evalS w@(While e s1) s = case (evalE e s) of ' (BoolVal True,s') -> let s' ' = evalS
s1 s' in evalS w s' ' (BoolVal False,s') -> s' _ -> error "条件必须是 BoolVal"
```

看到了吗？`evalS` 明确地接收其初始存储 `s`，并在 `evalE e s` 中显式使用它。生成的新存储在两个分支中都称为 `s'`。如果循环结束，则将 `s'` 作为最终存储返回。否则，`s'` 被用作循环体一次迭代的存储 `s1`，而由此产生的存储 `s''` 则用作下一次循环迭代的存储 `w`。

你的代码需要在每个求值阶段以同样明确的方式对存储进行命名并加以使用。让我们一步步来看。

```
evalS 跳过 s = 显示 s -- 我假设由于 Skip 返回一个空的字符串
```

你的假设是错误的。`evalS` 函数并不返回一个 `String`，无论是否为空：它返回的是一个 `Store`。那么，是哪个 `Store`？你的初始存储是 `s`：在执行“`skip`”之后的存储将如何与 `s` 相关？

```
evalS (Sequence s1 s2) s = evalS s1 >> evalS s2 -- 先执行 sequence1 然后执行 sequence2。我不是 qu
```

再一次，这是一个单子论方法，无法与这个问题相匹配。你需要将存储器，最初为  $s$ ，贯穿于顺序评估语句  $s_1$  和  $s_2$  的过程。“while” 案例提供了一个如何执行类似操作的好例子。

```
evalS (Expr e) s = ... Not sure what to do, here.
```

再者，“while” 示例展示了通过计算表达式来提取一个值和更新存储的一种方式。值得深思，不是吗？

```
evalS (If e s1 s2) s = ...
```

现在，“if” 通过评估一个条件开始，挺像 “while” 的，对吧？所以，我的建议总结起来就是：

- 暂时搁置 Monad 风格的代码，但在合适的时候再回头处理它；
- 阅读 “while” 的示例实现，理解它如何处理表达式和语句序列，并显式地传递存储；
- 使用类似的技术来实现其他结构。

出题的人已经好心地给了你一段代码，展示了你需要的所有内容。请通过理解并抓住提示来回报这份好意！

## 1.41 存在类型包装器的必要性

你的第一次尝试是使用存在类型的 *not*。而  $y$  我们的

```
列表 :: [(Int, 对于所有 a. 显示 a => Int -> a)]
```

要求第二个组件能够提供我选择的 *any* 可显示类型的元素，而不仅仅是您选择的 *some* 可显示类型。您正在寻找

```
列表 :: [(Int, 存在 a. 显示 a * (Int -> a))] -- 不是实际的 Haskell
```

但那并不是你所说的。数据类型打包方法允许你通过柯里化从 `forall` 中恢复 `exists`。你有

```
HRF :: 对所有 a. Show a => (Int -> a) -> HRF
```

这意味着要构建一个 HRF 值，你必须提供一个三元组，其中包含一个类型  $a$ 、一个针对  $a$  的 `Show` 字典，以及一个类型为  $\text{Int} \rightarrow a$  的函数。也就是说，HRF 构造器的类型实际上对这个非类型进行了柯里化。

```
HRF :: (存在 a. Show a * (Int -> a)) -> HRF -- 不是真正的 Haskell
```

你或许可以通过使用 `rank-n` 类型来对存在类型进行 Church 编码，从而避免数据类型方法

```
类型 HRF = 对所有 x. (对所有 a. Show a => (Int -> a) -> x) -> x
```

但那可能是过度了。

## 1.42 类型级函数中的非线性模式

Haskell 的类型级语言是一个纯粹的一级语言，其中“应用”只是另一个构造器，而不是一个计算的事物。确实存在绑定构造，如 `forall`，但类型级的相等性概念本质上仅仅是 $\alpha$ -等价：结构上只是绑定变量的重命名。实际上，我们整个构造器-类机制、单子等依赖于能够毫不含糊地拆解应用 `m v`。

类型级函数并不真正作为一等公民存在于类型级语言中：只有它们的完全应用才是如此。最终我们得到的是一种类型级表达式的等式理论（就的相等概念而言），其中约束被表达并得到求解，但这些表达式所指称的底层 *value* 概念始终是一阶的，因此始终可以赋予相等性。

因此，通过结构等式测试来解释重复的模式变量总是有意义的，这正是模式匹配在其1969年初版中设计的方式，作为对另一种语言的扩展，该语言根植于一种基本的第一价值观念——LISP。

## 1.43 玫瑰树的初始代数

我不鼓励谈论“哈斯克类别”，因为它会潜移默化地使你忽视在哈斯克编程中寻找其他类别结构。

确实，玫瑰树可以被看作是作用于“类型与函数”之上的一个自函子的一个不动点；这个范畴我们或许更适合称之为 `Type`，因为如今 `Type` 是类型之类型。若我们给自己配备一些常见的函子工具……

```
新类型 K a x = K a 派生 Functor -- 常量函子    新类型 f g x = P (f x, g x) 派生 Functor -- 笛卡尔积
```

... 和不动点 ...

```
新类型 FixF f = InF (f (FixF f))
```

然后我们可以取

```
类型 Rose a = FixF (P (K a) []) 模式 Node :: a -> [Rose a]
]-> Rose a 模式 Node a ars = InF (P (K a, ars))
```

`[]` 本身是递归的这一事实并不妨碍它在通过 `Fix` 构造递归数据类型时的使用。为了明确地说明递归，我们有了嵌套的固定点，这里选择了具有提示性的绑定变量名称：

```
玫瑰 a =  $\mu$ 玫瑰。 a * ( $\mu$ 列表。 1 + (玫瑰 * 列表))
```

现在，当我们到达第二个固定点时，我们有一个类型公式

```
1 + (玫瑰 * list)
```

它在玫瑰和列表中都是函子（实际上，是严格正的）。可以说它是一个双函子，但这并不是必需的术语：它是从  $(\text{Type}, \text{Type})$  到 `Type` 的函子。你可以通过在对偶的第二个分量上取一个固定点来构造一个 `Type -> Type` 函子，这正是上面发生的事情。

上述对 `Rose` 的定义失去了一个重要的性质。并不成立的是

```
Rose :: Type -> Type -- GHC 可能会这么说，但它在撒谎
```

仅仅是如果 `x :: Type`，那么 `Rose x :: Type`。特别是，

## 函子玫瑰

并不是一个类型良好的约束，这很遗憾，因为从直觉上讲，玫瑰树在其所存储的元素上应该具有函子性。

你可以通过将 Rose 构建为其自身作为一个双函子 (Bifunctor) 的不动点来解决这个问题。因此，实际上，当我们处理到列表时，作用域中已经有 *three* 个类型变量：a、rose 和 list，并且我们在它们全部上都具有函子性。你需要一个 *different* 的不动点类型构造器，以及一个用于构建 Bifunctor 实例的 *different* 工具集：对于 Rose 来说，事情会更容易一些，因为参数 a 并未在内部的不动点中使用；但在一般情况下，要将双函子定义为不动点就需要三函子，于是故事就这样展开了！

我的这个回答通过展示 *indexed* 类型在函子不动点构造下是 *closed* 的，从而说明了如何对抗这种泛滥。换言之，不要在 Type 中工作，而是在  $i \rightarrow \text{Type}$  中（针对索引类型 i 的完整多样性），这样你就已经为相互递归、GADT 等做好了准备。

所以，从宏观来看，玫瑰树是由互为不动点给出的，它们有一个完全合理的范畴解释，前提是你看清楚哪些范畴实际上在起作用。



## 第二章

# 模式匹配

### 2.1 用于对 ML 风格模式匹配进行类型检查的算法？

这是一个算法的草图。它也是Lennart Augustsson著名技术的基础，用于高效地编译模式匹配。（这篇论文在那个令人难以置信的FPCA会议录（LNCS 201）中，获得了很多引用。）其思想是通过反复将最一般的模式分解为构造函数案例，来重建一个详尽且不冗余的分析。

一般而言，问题在于你的程序拥有一组可能为空的‘实际’模式  $\{p_1, \dots, p_n\}$ ，而你想知道它们是否覆盖给定的‘理想’模式  $q$ 。作为起点，令  $q$  为变量  $x$ 。不变式在最初得到满足并在随后得以保持，即每个  $p_i$  都是  $\sigma_i q$ ，其中  $\sigma_i$  是一种将变量映射到模式的替换。

如何进行下去。如果  $n=0$ ，集合为空，那么你有一个未被模式覆盖的可能情况  $q$ 。抱怨  $p_s$  不是穷尽的。如果  $\sigma_1$  是变量的单射重命名，那么  $p_1$  捕获每个与  $q$  匹配的情况，所以我们是热的：如果  $n=1$ ，我们赢了；如果  $n>1$ ，那么哎呀， $p_2$  永远不会需要。否则，对于某个变量  $x$ ， $\sigma_1 x$  是一个构造器模式。在这种情况下，将问题拆分为多个子问题，每个子问题对应  $x$  类型的一个构造器  $c_j$ 。也就是说，将原始  $q$  拆分为多个理想模式  $q_j = [x := c_j y_1 \dots \text{arity}(c_j)]q$ ，并相应地为每个  $q_j$  细化模式，以保持不变量，丢弃那些不匹配的模式。

让我们以  $\{[], x :: y :: zs\}$  (为例，使用  $::$  作为 cons)。我们从开始

$xs$  覆盖  $\{[], x :: y :: zs\}$

并且我们有  $[xs := []]$ ，使得第一个模式成为该理想的一个实例。因此我们对  $xs$  进行拆分，得到

$[]$  覆盖  $\{[]\}$   $x :: ys$  覆盖  $\{x :: y :: zs\}$

第一个由空的单射重命名所证明，因此是可以的。第二个采用  $[x := x, ys := y :: zs]$ ，于是我们又继续，分裂  $ys$ ，得到。

$x :: []$  覆盖  $\{x :: y :: zs\}$  覆盖  $\{x :: y :: zs\}$

而且我们可以从第一个子问题看出，我们已经束手无策了。

重叠的情况更加微妙，并且允许变化，取决于您是否想标记任何重叠，或者仅仅是按从上到下的优先顺序完全冗余的模式。您的基本摇滚依然相同。例如，从以下开始

$xs$  covering  $\{[], ys\}$

使用`[xs := []]`来证明其中的第一个，因此进行分割。请注意，我们必须通过构造器案例来细化`ys`，以保持不变式。

`[]` 覆盖 `{[], []}` `x :: xs` 覆盖 `{y :: ys}`

显然，第一个情况严格来说是重叠。另一方面，当我们注意到需要细化一个实际的程序模式以保持不变量时，我们可以筛选出那些变得多余的严格细化，并检查至少有一个得以保留（如这里的`::`情况）。

因此，算法以一种受实际程序模式 `p` 启发的方式构建一组理想的穷举重叠模式 `q`。每当实际模式要求更详细的特定变量时，你将理想模式拆分为构造器案例。如果运气好，每个实际模式都被不相交的非空理想模式集覆盖，并且每个理想模式只被一个实际模式覆盖。生成理想模式的案例拆分树为你提供了高效的跳表驱动编译实际模式的方式。

我所展示的算法显然是终止的，但如果存在没有构造器的数据类型，它可能无法接受空模式集是完备的。这在依赖类型语言中是一个严重的问题，因为常规模式的完备性是不可判定的：合理的做法是允许“反驳”以及等式。在 Agda 中，您可以写 `()`，发音为“my Aunt Fanny”，在任何没有构造器细化的地方，这使您免于完成带返回值的等式的要求。通过添加足够的反驳，任何完备的模式集都可以变得 *recognizably* 完备。

总之，这就是基本情况。

## 2.2 Haskell 模式匹配

`Expr` 的数据类型声明会系统地生成一组模式，这些模式覆盖了类型为 `Expr` 的值所能 *be* 的所有可能情况。让我们来进行翻译

```
data Expr          -- any e :: Expr must be one of
= T                -- T
| Var Variable     -- (Var x)                -- where x :: Variable
| And Expr Expr    -- (And e1 e2)            -- where e1 :: Expr, e2 :: Expr
| Not Expr         -- (Not e1)               -- where e1 :: Expr
```

你可以看到，作为每个数据子句开头的 `T`、`Var`、`And` 和 `Not` 是构造子，并且存在于值语言中；而每个子句中的其余部分存在于类型语言中，用来说明 `Expr` 的每个组成部分必须具有的类型。每个对应的模式都由构造子应用于模式变量组成，这些模式变量代表具有给定类型的组件。基本上，出现在函数左侧的模式是通过反复将模式变量细化为其值可能采取的模式而构成的，*as indicated by their type*。

通过模式匹配编写函数并不是简单地说明要做什么`do`：而是说明对于输入`is`的所有可能情况，输出`is`应是什么。你需要将输入分析为几种情况，在这些情况下，你可以轻松地确定输出应当是什么。因此，从一个一般的情况开始。。。

`v :: Expr -> [变量] v e = 未定义`

...并加以完善。问“你能看出来它是什么了吗？”我们不能在不了解更多关于 `e` 的信息之前判断 `v e` 是什么。因此我们最好 *split* `e`。我们知道 `e :: Expr`，所以我们知道它的值可以匹配哪些模式。复制你的程序行四次，并在每一行中，将 `e` 替换为上述四种可能的模式之一。

$v :: \text{表达式} \rightarrow [\text{变量}] v$   $T = \text{未定义}$

$v(\text{变量 } x) = \text{未定义 } v$  (与  $e1\ e2$ ) =

未定义  $v$  (非  $e1$ ) = 未定义

现在，在每种情况下，你能告诉输出是什么吗？方便的是，你可以利用对组件的递归调用。假设你已经知道在尝试说明  $v(\text{And } e1\ e2)$  必须是什么时， $\text{vars } e1$  和  $\text{vars } e2$  的值。如果你按照正确的步骤进行操作，程序将是正确的。

我发现从具体例子来思考往往很有帮助。就拿你的测试示例来说。

$v(\text{非}(\text{且}(\text{变量 } "y")\ \text{真}))$

The 它应该是 provided 对吧？它符合哪种模式？(Tha"). Could you please provide the full text you'd like to have translated?

Not  $e1$  -- with  $e1 = \text{And } (\text{Var } "y")\ T$

什么是

$v\ e1$

看起来，它最好是

$["y"]$

在这个例子中， $v$ （不是  $e1$ ）如何用  $v\ e1$  来表示？完全相同。这可能暗示一个合适的表达式来替换未定义的部分。

$v(\text{Not } e1) = \text{未定义}$  —— 你现在能看出这是什么吗？

当然，一个具有启发性的例子只是一个良好的开始，并不能保证正确性。

主要信息：(1) 通过拆分模式变量来构建模式，通过查看类型声明来找出可能的模式；(2) 假设对组件的递归调用会给出正确的答案，然后尝试为整个问题构造正确的答案。

*Shameless plug: shplit is a simple tool I built for my students, capturing message (1) mechanically.*

## 2.3 复杂模式匹配

当我还是80年代的小男孩时，我实现了一种具有复杂模式的函数式语言，采用了那种风格。它相当于允许在模式中使用  $++$ 。最终的模式是模糊的，因此匹配过程涉及回溯搜索：程序员可以有效地指定是最小化还是最大化匹配  $++$  左侧前缀的长度。该语言具有一种“模式守卫”形式，以便可以测试候选匹配，查看后续计算是否成功，失败时则被拒绝。最终的程序通常直观明了，易于理解其含义。这非常有趣。

这些天，当面对这些问题时，我会使用 `span`，如果这样还不够，我会使用解析器组合器。

$\text{span} :: (a \rightarrow \text{布尔}) \rightarrow [a] \rightarrow ([a], [a])$

`span`，应用于谓词  $p$  和列表  $xs$ ，返回一个元组，其中第一个元素是满足  $p$  的  $xs$  元素的最长前缀（可能为空），第二个元素是列表的其余部分。

因此，特别是 `span (/= ',)` 将把字符串拆分为第一个逗号之前的部分（如果没有逗号，则为整个字符串），以及其余部分（如果有逗号，则从逗号开始）。

如果那样还不行，我就使用解析器组合器。

但我总是记得以前是多么容易。

## 2.4 如何返回我进入之前的一个元素？

在评论中可以清楚地观察到，你的 `otherwise`（这只是 `True` 的同义词）需要像其他守卫一样带有 `=` 符号，但我会做一些其他调整。

部分函数 `head` 和 `tail` 可能最好避免使用，特别是因为有一种通过模式匹配解决这个问题的好方法。

```
elementBefore :: Eq a => a -> [a] -> Maybe a
elementBefore elt (x : xs@(y : _)) | y == elt = Just x | otherwise = elementBefore elt xs
elementBefore _ _ = Nothing
```

关键在于使用 `@` 来构造一个“as-pattern”，它同时为列表 `xs` 的尾部命名（以便在我们不走运时使用），并将其匹配为 `(y :)`（这样我们就能看出是否已经赢了）。

当我还是个孩子的时候，我和父亲大概会写出类似这样的东西

```
元素之前 元素 ( _ ++ x : 元素 : _ ) = 仅 x 元素之前 _ _ = 无
```

但那一直都过于简单，称不上是合法的 Haskell。

## 2.5 猛禽火箭筒变焦

该规范并不完全清楚，但听起来你是想收集输入中所有出现在某个“Z”之后第三个位置上的字符，因此从

秃鹰火箭筒变焦

我们得到

RDKM

如果对问题的表述不够清晰，就很难给出精确的建议。但我希望你能帮助你克服一些小的困扰，从而让你能够专注于问题本身的逻辑。

让我们从类型开始。你有

```
someFun :: 字符串 => 字符串 -> 字符串 -> 字符串
```

但是在 `=>` 的左边是放置 *properties* 类型表达式的位置，通常涉及可以代表许多类型的变量，例如 `Eq a`（意思是无论 `a` 是什么类型，我们都可以测试相等性）。`String` 是一个类型，而不是一个属性，因此它不能放在 `=>` 的左边。把它去掉。这样得到

```
someFun :: String -- 输入 -> String -- 累积输出 (?) -> String -- 输出
```

目前不清楚您是否真的需要一个累加器。假设您知道输出为

```
"ZARD BAZOOKA BOOM" -- "DKM", right?
```

你能为其计算输出吗

ZZARD火箭筒轰炸 -- RDKM

? 只是在前面多了一个“R”，对吧？你在使用尾递归来`do`下一步，而通常更简单的是去思考事物应该如何`be`。如果你知道列表尾部的输出`is`，那么就说出整个列表的输出`is`。为什么不直接把输入映射到输出呢，于是

```
someFun :: 字符串 -> 字符串
```

现在，从最简单的模式开始进行模式匹配。

```
someFun s = 未定义
```

你能从输入中看到足够的信息来确定输出吗？显然不能。输入是空的还是有一个首字符是有区别的。分成两种情况。

```
someFun "" = undefined someFun (c : s) = undefined -- c 是第一个字符，s 是字符串的其余部分
```

它是否以‘Z’开头也很重要。请小心使用 *single* 引号来表示 Char，使用 *double* 引号来表示 String：它们是不同的类型。

```
一些有趣的 "" = 未定义 一些有趣的 ('Z' : s) = 未定义 -- 第一个字符是 Z
```

```
一些有趣的 (c : s) = 未定义
```

在‘Z’的情况下，你还需要确保`s`至少有三个字符，我们关注的是第三个字符，因此

```
一些有趣的 "" = 未定义 -- 输入为空 一些有趣的 ('Z' : s@(_ : _ : d : _)) = 未定义 -- 首先是 'Z' 且 d 为 3 后
```

```
一些有趣的 (c : s) = 未定义 -- 输入非空
```

@ 是一种“as 模式”，它允许我为整个尾部 `s` 命名，同时检查它是否匹配 `(:: d :)`，并获取 ‘Z’ 之后的第三个字符。

到目前为止，我还没有考虑输出，只是关注我需要从输入中看到什么。让我们弄清楚输出必须是什么。在第一种情况下，空输入得到空输出

```
someFun "" = ""
someFun ('Z' : s@(_ : _ : d : _)) = undefined -- first is 'Z' and d is 3 later
someFun (c : s) = undefined -- input nonempty
```

在另外两种情况下，我们可以假设 `someFun s` 已经告诉我们列表尾部的输出，因此我们只需要弄清楚如何完成整个列表的输出。在最后一行，尾部的输出正是我们想要的。

```
someFun "" = ""
someFun ('Z' : s@(_ : _ : d : _)) = undefined -- first is 'Z' and d is 3 later
someFun (c : s) = someFun s
```

但在我们发现 `d` 位于初始“Z”之后第三个位置的情况下，我们需要确保 `d` 位于输出的开头。

```
someFun "" = "" someFun ('Z' : s@(_ : _ : d : _)) = d : someFun s
someFun (c : s) = someFun s
```

只是检查：

```
*Main> someFun "BUZZARD BAZOOKA ZOOM" "
RDKM"
```

关键思想是弄清楚如何根据各个部分的输出来表达整个输入的输出：是什么 *it is*，而不是 *to do*。在这里，你可以假设尾部 `s` 的输出已正确计算，因此你只需要弄清楚是否需要返回任何额外的内容。

## 2.6 为什么 ++ 在模式匹配中不被允许？

这是一个值得提问的问题，到目前为止，已经得到了合理的回答（只是构造函数被允许，提到单射性，提到歧义），但仍然有时间改变这一切。

我们可以说明规则是什么，但关于为什么规则是这样的多数解释，都是从对问题的过度泛化开始的，转而去解释为什么我们不能对任意老函数进行模式匹配（咕哝一句 Prolog）。这忽略了一个事实：++ 并不是随便什么函数；它是一个（在空间上）*linear* 的把东西拼接在一起的函数，由列表的拉链结构所诱导。模式匹配关乎把东西拆开，确实如此，并且用把它们拼接起来的器件以及代表各个组件的模式变量来记号化这一过程。它的动机在于清晰性。所以我要

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
lookup k (_ ++ [(k, v)] ++ _) = Just v
lookup _ _ = Nothing
```

不仅因为它会让我想起三十年前，当我实现了一种功能性语言时，它的模式匹配正好提供了这个。

认为它存在歧义的反对意见是合理的，但并非致命问题。像 ++ 这样的拼接器只会对有限输入给出有限多种分解（如果你在处理无限数据，那只能自求多福），因此其中所涉及的，充其量只是 *search*，而不是 *magic*（发明任意的输入——这些输入可能已被任意的函数丢弃）。搜索需要某种优先级机制，但我们的有序匹配规则同样如此。搜索也可能导致失败，但匹配同样也可能失败。

我们有一种合理的方式来管理计算，通过 *Alternative* 抽象提供备选方案（失败和选择），但我们并不习惯将模式匹配视为这种计算的一种形式，这就是为什么我们仅在 *expression* 语言中利用 *Alternative* 结构的原因。一个高尚的，尽管是异想天开的例外是 *do-notation* 中的匹配失败，它调用相关的 *fail* 而不是必然崩溃。模式匹配是尝试计算适合评估“右侧”表达式的环境；计算此环境的失败已经被处理了，那为什么不选择呢？

编辑：我当然应该补充一下，只有在模式中有多个可伸缩的元素时，才真的需要搜索，因此提议的 *xs++[x]* 模式不应该触发任何选择。当然，找到列表的末尾需要时间。

设想有一种用于书写 *Alternative* 计算的有趣括号，例如， $(|)$  表示空， $(|a_1|a_2|)$  表示  $(|a_1|) <|> (|a_2|)$ ，而普通的老式  $(|f\ s_1 \dots s_n|)$  表示纯粹的  $f <*> s_1 \dots <*> s_n$ 。人们也完全可以想象， $(|case\ a\ of\ \{p_1 \rightarrow a_1; \dots p_n \rightarrow a_n\}|)$  会以 *Alternative* 组合子的方式，对搜索模式（例如涉及 ++）进行一种合理的翻译。我们可以写成

```
lookup :: (Eq k, Alternative a) => k -> [(k, v)] -> a k
lookup k xs = (|case xs of _ ++ [(k, v)] ++ _ -> pure v|)
```

我们可以获得任何由可微函子的不动点生成的 *datatype* 的合理搜索模式语言：符号微分正是将结构的元组转化为可能子结构的选择。老式的 ++ 就是子列表的列表表示例（这令人困惑，因为一个有孔的子列表的列表看起来很像是一个列表，但对于其他数据类型来说情况并非如此）。

令人啼笑皆非的是，只需一点 *LinearTypes*，我们甚至可以既通过它们的孔洞也通过它们的根来抓住那些多孔数据，然后以常数时间进行破坏式地填补。只有当你没有注意到自己正在这么做时，这才算是骇人听闻的行为。

## 第3章

# 递归

### 3.1 什么是参数递归？

是的，这是para。与catamorphism或fold进行比较。 r:

```
para :: (a -> [a] -> b -> b) -> b -> [a] -> b foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
para c n (x : xs) = c x xs (para c n xs) foldr c n (x : xs) = c x (foldr c n xs)
para c n [] = n foldr c n [] = n
```

有些人将参数变换称为“原始递归”，与迭代（foldr）相对，后者被称为“迭代”。

foldr 的两个参数分别为输入数据的每个递归子对象（这里是列表的尾部）提供一个递归计算的值，而 para 的参数则同时获得原始子对象和从中递归计算得到的值。

一个用 para 能很好地表达的示例函数是：一个列表的真后缀集合。

```
suff :: [x] -> [[x]] suff = para (\ x xs suffxs -> xs : suffxs) []
```

以便

```
后缀 "suffix" = ["uffix", "ffix", "fix", "ix", "x", ""]
```

可能更简单的是

```
safeTail :: [x] -> Maybe [x] safeTail = para (\ _ xs _ -> Just xs) Nothing
```

其中“cons”分支会忽略其递归计算得到的参数，只是返回尾部。在惰性求值下，递归计算根本不会发生，尾部可以以常数时间被提取。

你可以很容易地使用 para 定义 foldr；从 foldr 定义 para 稍微复杂一些，但绝对是可能的，每个人都应该知道它是如何做的！

```
foldr c n = para (\ x xs t -> c x xs t) n
para c n = snd . foldr (\ x (xs, t) -> (x : xs, c x xs t)) ([], n)
```

定义 `para` 使用 `foldr` 的诀窍是重建一个 *copy* 的原始数据，这样我们在每一步都可以访问尾部的副本，即使我们无法访问原始数据。最后，`snd` 丢弃输入的副本，仅返回输出值。这并不是很高效率，但如果你对纯粹的表达能力感兴趣，`para` 给出的效果并不比 `foldr` 更强。如果你使用这种 `foldr` 编码的 `para` 版本，那么 `safeTail` 最终将需要线性时间，逐个元素地复制尾部。

所以，就是这样：`para` 是 `foldr` 的一个更方便的版本，它让你能够立即访问列表的尾部，以及从该尾部计算得到的值。

在一般情况下，处理作为函子递归不动点生成的数据类型

数据 `Fix f = 在 (f (Fix f))`

你有

```
cata :: 函子 f => (f t -> t) -> Fix f -> t
para :: 函子 f => (f (Fix f, t) -> t) ->
Fix f -> t
```

```
cata phi (In ff) = phi (fmap (cata phi) ff)
para psi (In ff) = psi (fmap keep
Copy ff) 其中 keepCopy x = (x, para psi x)
```

而且，这两者是相互可定义的，其中 `para` 可以通过同样的“复制一份”的技巧从 `cata` 定义出来

```
para psi = snd . cata (\ fxt -> (In (fmap fst fxt), psi fxt))
```

再者，`para` 并不比 `cata` 更具表现力，但如果你需要方便地访问输入的子结构，它更为便捷。

编辑：我想起了另一个不错的例子。

考虑由 `Fix TreeF` 给定的二叉搜索树，其中

数据 `TreeF 子 = 叶 | 节点 子 整数 s` ub

并尝试为二叉搜索树定义插入操作，先用 `cata`，再用 `para`。你会发现 `para` 版本要容易得多，因为在每个节点你只需要在一个子树中插入，同时保持另一个子树原样不变。

## 3.2 为什么可以使用 `foldl` 反转列表，而不能使用 `foldr` 在 Haskell 中反转列表

每个 `foldl` 都是一个 `foldr`。

让我们回顾这些定义。

```
foldr :: (a -> s -> s) -> s -> [a] -> s
foldr f s [] = s
foldr f s (a:as) = f a (foldr f s as)
```

这是标准的单步迭代器，用于列表。我曾经让我的学生们拍打桌子并 chant：“你对空列表做什么？你对：作为做什么？”这就是你弄清楚 `s` 和 `f` 分别是什麼的方式。

如果你仔细想想正在发生的事情，你会发现 `foldr` 实际上计算了一个由 `f a` 函数组成的大型组合，然后将该组合应用到 `s` 上。

```
foldr f s [1, 2, 3]
= f 1 . f 2 . f 3 . id $ s
```

### 3.2. WHY CAN YOU REVERSE LIST WITH FOLDL, BUT NOT WITH FOLDR IN HASKELL 57

现在，让我们来看看 foldl

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t [] = t
foldl g t (a : as) = foldl g (g t a) as
```

那也是对列表的一次单步迭代，但带有一个会随着迭代而变化的累加器。让我们把它移到最后，这样列表参数左侧的所有内容都保持不变。

```
翻转 . foldl :: (t -> a -> t) -> [a] -> t -> t
翻转 (foldl g) [] t = t
翻转 (foldl g) (a : as) t = 翻转 (foldl g) as (g t a)
```

现在，如果我们将 = 向左移动一个位置，就可以看到一步迭代。

```
翻转 . foldl :: (t -> a -> t) -> [a] -> t -> t
翻转 (foldl g) [] = \ t -> t
翻转 (foldl g) (a : as) = \ t -> 翻转 (foldl g) as (g t a)
```

在每种情况下，我们计算 *what we would do if we knew the accumulator*，并用 \ t -> 进行抽象。对于 []，我们将返回 t。对于 a : as，我们将使用 g t a 作为累加器处理尾部。

但现在我们可以将 flip (foldl g) 转换成一个 foldr。将递归调用抽象出来。

```
翻转 . foldl :: (t -> a -> t) -> [a] -> t -> t
翻转 (foldl g) [] = \ t -> t
翻转 (foldl g) (a : as) = \ t -> s (g t a) 其中 s = 翻转 (foldl g) as
```

现在我们可以将其转换为一个 foldr，其中类型 s 被实例化为 t -> t。

```
翻转 . foldl :: (t -> a -> t) -> [a] -> t -> t
翻转 (foldl g) = foldr (\ a s -> \ t -> s (g t a)) (\ t -> t)
```

所以 s 说“a 会怎么做累加器”，我们给回 \ t -> s (g t a)，这就是“a : as 会怎么做累加器”。翻回来。

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g = flip (foldr (\ a s -> \ t -> s (g t a)) (\ t -> t))
```

Eta-展开.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t 作为 = flip (foldr (\ a s -> \ t -> s (g t a)) (\ t -> t)) t 作为
```

减少翻转。

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = foldr (\ a s -> \ t -> s (g t a)) (\ t -> t) as t
```

所以我们计算“如果我们知道累加器会怎么做”，然后将初始累加器输入进去。

将其高尔夫球略微压低是适度的指导。我们可以去掉 \ t ->。

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t 作为 = foldr (\ a s -> s . ( 'g ' a )) id 作为 t
```

现在让我使用来自 Control.Arrow 的 >>> 来反转那个组合。

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = foldr (\ a s -> ( 'g ' a ) >>> s) id as t
```

也就是说, `foldl` 计算一个大的 *reverse* 组合。因此, 例如, 给定 `[1,2,3]`, 我们得到

```
foldr (\ a s -> ( 'g ' a ) >>> s) id [1,2,3] t = ( ( 'g ' 1
) >>> ( 'g ' 2 ) >>> ( 'g ' 3 ) >>> id ) t
```

其中“管道”从左侧传入其参数, 因此我们得到

```
(( 'g ' 1 ) >>> ( 'g ' 2 ) >>> ( 'g ' 3 ) >>> id) t
= ( ( 'g ' 2 ) >>> ( 'g ' 3 ) >>> id ) (g t 1) = ( ( 'g ' 3 ) >>> id ) (g (g t 1) 2) = id (g (g (g t 1) 2) 3) = g (g (g t 1) 2) 3
```

并且如果你取 `g =` 的 `flip (:)` 和 `t = []`, 你会得到

```
flip (:) (flip (:) (flip (:) [] 1) 2) 3 = flip (:) (flip (:) (1 : []) 2) 3
= flip (:) (2 : 1 : []) 3 = 3 : 2 : 1 : [] = [3, 2, 1]
```

也就是说,

逆向为 `= foldr (\ a s -> > (a :) >>> s) id` 作为 `[]`

通过实例化 *general* 的 `foldl` 转换为 `foldr`。

仅限受虐狂。执行 `cabal install newtype` 并导入 `Data.Monoid`、`Data.Foldable` 和 `Control.Newtype`。添加那个悲剧性缺失的实例:

```
实例 Newtype (Dual o) o 其中 pack = Dual unpa
ck = getDual
```

注意到, 一方面, 我们可以通过 `foldr` 来实现 `foldMap`

```
foldMap :: 单位元素 x => (a -> x) -> [a] -> x
foldMap f = foldr (mappend . f) mempty
```

但反之亦然

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = flip (ala' Endo foldMap f)
```

这样 `foldr` 就在由自函数复合构成的么半群中进行累积, 但现在为了得到 `foldl`, 我们让 `foldMap` 在 `Dual` 么半群中工作。

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl g = flip (ala' Endo (ala' Dual foldMap f) (flip g))
```

`Dual (Endo b)` 的 `mappend` 是什么? 在忽略包装 (*modulo wrapping*) 的情况下, 它恰好是反向复合, `>>>`。

### 3.3 fold 能否用于创建无限列表？

foldl 和 foldr 函数是列表-*consumers*。正如 svenningsson 的回答正确指出的那样，unfoldr 是一种列表-*producer*，适合捕捉 fibs 的 co-递归结构。

然而，鉴于 foldl 和 foldr 在其返回类型上是多态的，也就是说，它们通过消费一个列表所产生的结果是多态的，因此有理由提出这样一个问题：它们是否可以被用来消费一个列表并生成另一个列表？这些生成的列表中是否有可能是无限的？

查看 foldl 的定义

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (b : bs) = foldl f (f a b) bs
```

我们看到，要让 foldl 产生任何结果，它所消费的列表必须是有限的。因此，如果 foldl f a 产生了无限的输出，那要么是因为 a 是无限的，要么是因为 f 有时会进行无限的列表生成。

对于 foldr，情况就不同了

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a [] = a
foldr f a (b : bs) = f b (foldr f a bs)
```

这承认了一种惰性的可能性：f 可能会为从输入中消耗的每个 b 生成一些输出。诸如这样的操作

```
map g = foldr (\ b gbs -> g b : gbs) [] -- 代码高尔夫玩家更喜欢 ((:) . g)stutter = foldr (\ x xxs -> x : x : xxs) []
```

对每个输入只产生一点输出，却能从无限输入中产生无限输出。

因此，一个耍巧的人可以将任何无穷递归表达为在一个无限列表上的非递归 foldr。例如，

```
foldr (\ _ fibs -> 1 : 1 : zipWith (+) fibs (tail fibs)) undefined [1..]
```

（编辑：或者说，甚至

```
foldr (\ _ fib a b -> a : fib b (a + b)) undefined [1..] 1 1
```

这更接近题目中的定义。）尽管这一观察是正确的，但几乎不能表明这是一种健康的编程风格。

### 3.4 我如何为一个用于一般递归方案的datatype提供一个Functor实例？

这对我来说是个老伤疤。关键点在于你的 ExprF 在 both 其参数上是函子性的。因此如果我们有

类 Bifunctor b 其中

```
双映射 :: (x1 -> y1) -> (x2 -> y2) -> b x1 x2 -> b y1 y2
```

然后你可以定义（或者设想由一台机器为你定义）

```
实例 Bifunctor ExprF, 其中 bimap k1 k2 (Val a) = Val (k1 a) bi
map k1 k2 (Add x y) = Add (k2 x) (k2 y)
```

现在你可以拥有

新类型 `Fix2 b a = MkFix2 (b a (Fix2 b a))`

伴随

```
map1cata2 :: Bifunctor b => (a -> a') -> (b a' t -> t) -> Fix2 b a -> t
map1cata2 e f (MkFix2 bar) = f (bimap e (map1cata2 e f) bar)
```

这反过来意味着，当你在其中一个参数中取一个不动点时，剩下的部分在另一个参数中仍然是函子的。

实例 `Bifunctor b => Functor (Fix2 b)` 其中 `fmap k = map1cata2 k MkFix2`

和你有点像得到了你想要的东西。但是你的 `Bifunctor` 实例不是凭空构建的。并且你需要一个不同的固定点操作符和一种全新的 `functor`，这有点烦人。问题是你现在有 *two* 种子结构：“值”和“子表达式”。

这是转折。存在一个在固定点下的函子概念 *closed*。打开厨房水槽（特别是 `DataKinds`）并

类型 `s :-> t = 对于所有 x. s x -> t x`

类 `FunctorIx (f :: (i -> *) -> (o -> *))` 其中 `mapIx :: (s :-> t) -> f s :-> f t`

请注意，“元素”属于一种按 *i* 索引的类型，而“结构”属于按其他类型 *o* 索引的类型。我们将元素上的 *i* 保持函数视为结构上的 *o* 保持函数。关键是，*i* 和 *o* 可以是不同的。

魔法词是“1, 2, 4, 8, 是时候做指数运算了！”。一种类型 *\** 可以很容易地转化为一个平凡索引的 GADT，类型是 `() -> *`。两个类型可以合并在一起，形成一个 GADT，类型是 `Either () () -> *`。这意味着我们可以将两种种子结构合并在一起。一般来说，我们有一种类型级的 `Either`。

数据 案例 `:: (a -> *) -> (b -> *) -> 或者 a b -> *` 其中 `CL :: f a -> 案例 f g (左 a) C`  
`R :: g b -> 案例 f g (右 b)`

配备了其“地图”概念

```
mapCase :: (f :-> f') -> (g :-> g') -> Case f g :-> Case f' g'
mapCase ff gg (CL fx) = CL (ff fx) mapCase ff gg (CR gx) = CR (gg gx)
```

因此我们可以将我们的双因子重新函子化为以 `Either` 为索引的 `FunctorIx` 实例。

现在我们可以取任何节点结构 *f* 的不动点，其中包含元素 *p* 或子节点的位置。这个过程与我们之前的处理完全相同。

新类型 `FixIx (f :: (Either i o -> *) -> (o -> *)) (p :: i -> *) (b :: o) = MkFixIx (f (Case p (FixIx f p)) b)`

`mapCata :: forall f p q t. FunctorIx f =>`

```
(p :-> q) -> (f (案例 q t) :-> t) -> FixIx f p :-> t
mapCata e f (MkFixIx node) = f (mapIx (mapCase e (mapCata e f)) node)
```

但现在，我们得出一个事实：FunctorIx 对 FixIx 是封闭的。

实例 FunctorIx f => FunctorIx (FixIx f), 其中 mapIx f = mapCata f Mk  
FixIx

索引集上的函子（具有额外的、可以改变索引的自由度）可以非常精确，也非常强大。它们享有比普通函子多得多的便利闭包性质。我不认为它们会流行起来。

### 3.5 Haskell 中有 (术语变换) 同态吗？

供参考，以下是术语。...

数据 项 a = 变量 a | Lambda a (项 a) |  
应用 (项 a) (项 a)

（我注意到变量的表示是抽象化的，这通常是一个不错的做法。）……而这里是所提出的函数。

```
fmap' :: (Term a -> Term a) -> Term a -> Term a
fmap' f (Var v) = f (Var v) f
fmap' f (Lambda v t) = Lambda v (fmap' f t)
fmap' f (Apply t1 t2) = Apply (fmap' f t1) (fmap' f t2)
```

让我困扰的是，这一定义中 f 仅仅应用于形式为 (Var v) 的项，因此你完全可以实现 *substitution*。

```
替换 :: (a -> 项 a) -> 项 a -> 项 a
替换 f (变量 v) = f v
替换 f (λ v t) = λ v (替换 f t)
替换 f (应用 t1 t2) = 应用 (替换 f t1) (替换 f t2)
```

如果你在区分约束变量和自由变量时稍微更仔细一些，你就能够让 Term 成为一个 Monad，并由替换来实现 (>>=)。一般来说，项可以具有用于重命名的 Functor 结构，以及用于替换的 Monad 结构。Bird 和 Paterson 有一篇很精彩的论文讨论了这一点，不过我扯远了。

与此同时，如果你<sup>do</sup>想在变量之外进行操作，一种常见的方法是使用通用遍历工具包，如 augustss 所建议的 uniplate。另一种可能性，也许更具纪律性，是使用适合你类型的 ‘fold’。

```
tmFold :: (x -> y) -> (x -> y -> y) -> (y -> y -> y) -> 项 x -> y
tmFold v l a (Var x) = v x
tmFold v l a (Lambda x t) = l x (tmFold v l a t)
tmFold v l a (Apply t t') = a (tmFold v l a t) (tmFold v l a t')
```

在这里，v、l 和 a 定义了一个替代 *algebra* 的术语形成操作，仅作用于 y，而不是术语 x，解释如何处理变量、lambda 和应用。你可以选择将 y 设为 m (Term x)，其中 m 是某个合适的单子（例如，用于变量的环境传递），而不仅仅是 Term x 本身。每个子项都会被处理以生成 y，然后选择合适的函数来生成整个术语的 y。fold 捕捉了标准的递归模式。

普通的一阶数据类型（以及一些良好行为的高阶数据类型）都可以配备折叠算子。以牺牲可读性为代价，你甚至可以一劳永逸地写出这个折叠算子。

数据  $\text{Fix } f = \text{在 } (f (\text{Fix } f))$

```
fixFold :: Functor f => (f y -> y) -> Fix f -> y
fixFold g (In xf) = g (f
map (fixFold g) xf)
```

数据  $\text{TermF } a \ t = \text{Var}$

$F \ a \mid \text{LambdaF } a \ t \mid$

$\text{ApplyF } t \ t$

$\text{type Term } a = \text{Fix } (\text{TermF } a)$

不同于你的递归  $\text{Term } a$ ，这个  $\text{TermF } a \ t$  说明了如何构造一个项的 *one* 层，在子项位置放置  $t$  个元素。通过使用递归的  $\text{Fix}$  类型，我们取回递归的  $\text{Term}$  结构。在外观上我们会稍微损失一些，因为每一层都多了一层  $\text{In}$  的包裹。我们可以定义

```
var x = 在 (VarF x)
lambda x t = 在 (Lambd
aF x t)
apply t t' = 在 (Apply x t t')
```

但是我们不能在模式匹配中使用这些定义。好处是，我们可以在没有额外成本的情况下使用通用的  $\text{fixFold}$ 。为了从一个项计算  $y$ ，我们只需要提供一个类型为  $\text{Fix } f \rightarrow y$  的函数

$\text{TermF } a \ y \rightarrow y$

它（就像上面的  $v$ 、 $l$  和  $a$  一样）说明了如何处理任何其子项已经被处理为  $y$  类型值的项。通过在类型中明确一层由什么组成，我们可以利用逐层工作的通用模式。

### 3.6 这个斐波那契数列函数是递归的吗？

哎呀，这真是一团术语上细微区分的乱麻。什么是“这个”？

```
fib=0:1:zipWith (+) fib (tail fib)
```

这不是一个递归函数。它也不是递归数据。它是一个递归定义。正在被定义的是什么？

$\text{fib}$

根据这个定义， $\text{fib}$  是一种什么类型的东西？

[整数]

一列整数（或者可能是一列任何数字的东西）。

$\text{fib}$  是一个函数吗？不是，它是一个列表。 $\text{fib}$  是递归定义的吗？是的。如果我们用同类型的非递归函数替换  $\text{zipWith}$ （例如  $\backslash f \ x \ y \ s \rightarrow x \ s$ ）， $\text{fib}$  还会是递归定义的吗？是的，尽管它将是一个不同的递归定义的列表。

$\text{fib}$  是一个循环列表吗？不是。“递归数据结构”是否意味着“循环数据结构”？根据 Hoare 的论文《Recursive Data Structures》，并非如此：[http://portal.acm.org/book\\_gateway.cfm?id=63445&type=pdf&\\_p217-hoare.pdf&coll=&dl=&CFID=15151515&CFTOKEN=6184618](http://portal.acm.org/book_gateway.cfm?id=63445&type=pdf&_p217-hoare.pdf&coll=&dl=&CFID=15151515&CFTOKEN=6184618)

在类型化的语境中，“递归数据结构”不多不少就是“递归定义的类型的一个居留者（值）”。相应地，“ $\text{fib}$ ”是一个递归数据结构，尽管它并非递归定义的，而且确实可以被诸如  $++$  这样的递归函数所处理。

短语“递归函数”意指“以递归方式定义的函数”。短语“递归值”意指“以递归方式定义的值”，例如存在于非严格语言中：而严格语言存在“值递归”问题。

如果你觉得这太吹毛求疵了，试着在一种 *total* 编程语言中用这种方式来定义 `fib`，你就会发现，“递归定义”的概念会分裂为“通过结构递归的定义”（以一种会停止的方式消耗数据）和“通过受保护的共递归的定义”（以一种会继续前进的方式生成数据），而 `fib` 属于后者。在那种设定下，`fib` 的生产性关键取决于 `zipWith` 的惰性。当然，在 Haskell 的语境中，你并不需要操心这些东西来判断某个定义属于哪一类，只需要判断它是否有哪怕一半的机会真的能工作。

### 3.7 有人能解释一下这个惰性斐波那契解法吗？

你的“Because”并没有讲完整个故事。你把列表截断在“the story so far”处，并采用急切求值来进行评估，然后又疑惑剩下的部分是从哪里来的。这还不足以理解真正发生了什么，所以这是个好问题。

当你做出这个定义时，计算了什么

```
fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)      s)
```

? 很少。一旦你开始 *use* 这个列表，计算就会开始。惰性计算只在需要时才发生。

什么是需求？你可以问“你是 `[]` 还是 `x : xs`？” ，如果是后者，你就能把握住各个部分。

当我们问关于虚假的问题时，我们得到的是

```
fibs = x0 : xs0
x0 = 0
xs0 = 1 : zipWith (+) fibs (drop 1 fibs)
)
```

但这意味着（替换 `fibs` 然后再替换 `x0`）

```
xs0 = 1 : zipWith (+) (0 : xs0) (drop 1 (0 : xs0))
```

而当我们再次询问时，我们得到的就是那个

```
xs0 = x1 : xs1
x1 = 1
xs1 = zipWith (+) (0 : xs0) (drop 1 (0 : xs0))
```

所以

```
xs1 = zipWith (+) (0 : 1 : xs1) (drop 1 (0 : 1 : xs1))
```

但现在事情变得有趣了，因为我们必须做一些工作。注意，只需要做足以回答问题的工作。当我们查看 `xs1` 时，我们会强制执行 `zipWith`，而这又会强制执行 `drop`。

```
xs1 = zipWith (+) (0 : 1 : xs1) (drop 1 (0 : 1 : xs1)) = zipWith (+) (0 : 1 : xs1) (1 : xs1)
      = (0 + 1) : zipWith (+) (1 : xs1) xs1
```

所以

```
xs1 = x2 : xs2
x2 = 0 + 1 = 1
xs2 = zipWith (+) (1 : xs1) xs1
      = zipWith (+) (1 : 1 : xs2) (1 : xs2)
```

看到了吗？我们一直认为，我们仍然知道一个 `zip` 列表的前两个元素，以及另一个列表的第一个元素。这意味着我们将能够生成下一个输出 *and*，并刷新我们的“缓冲区”。当我们查看 `xs2` 时，我们得到

```
xs2 = zipWith (+) (1 : 1 : xs2) (1 : xs2) = (1 + 1) : zipWith
(1 : xs2) xs2 xs2 = x3 : xs3 x3 = 1 + 1 = 2 xs3 = zipWith (1 :
xs2) xs2 = zipWith (1 : 2 : xs3) (2 : xs3)
```

而且我们又一切就绪了！

每当我们请求下一个元素时，我们也就离 `zipWith` 用尽元素更远了一步；这正是好事，而且恰逢其时。

使得值能在恰到好处时刻出现的那些规约，没有任何一部分体现在类型之中。目前，这需要程序员来确保类型良好的程序在提出需求时不会因为数据耗尽而出错。（我确实有计划对此做点什么，但这里就不展开了。）

关键在于，惰性、“按需”的计算意味着我们 *don't* 必须将列表截断为在过程开始时我们能够看到的那些元素。我们只需要知道我们始终可以迈出下一步。

### 3.8 不动点上的幺半折叠

这是一个解决方案的要点。我已经开启了

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable, PatternSynonyms #-}
```

让我们回顾一下不动点和大范畴映射。

```
newtype Fix f = In {out :: f (Fix f)}
```

```
cata :: Functor f => (f t -> t) -> Fix f -> t
cata alg = alg . fmap (cata alg) . out
```

代数运算，`alg :: f t -> t`，接受一个节点，其中子节点已经被 `t` 值替代，然后返回父节点的 `t` 值。`cata` 操作符通过解包父节点、递归处理所有子节点，然后应用 `alg` 完成任务。

所以，如想要在这样的结构中计数叶子，我们可以开始 `li` `ke` 这个：

```
leaves :: (Foldable f, Functor f) => Fix f -> Integer
leaves = cata sumOrOne w
here -- sumOrOne :: f Integer -> Integer
```

这个代数 `sumOrOne` 可以查看父节点的每个子节点中的叶子数量。由于 `f` 是一个 `Functor`，我们可以使用 `cata`。而且因为 `f` 是 `Foldable`，我们可以计算子节点中叶子的总数量。

`sumOrOne fl = 情况 sum fl 的`

...

然后有两种可能：如果父节点没有子节点，它的叶子和将是 0，我们可以检测到，但这意味着父节点本身是叶子，因此应返回 1。否则，叶子和将为非零，在这种情况下父节点不是叶子，因此它的叶子和确实是其子节点叶子和的总和。这样我们得到

`leaves :: (Foldable f, Functor f) => Fix f -> Integer` `leaves = cata sumOrOne` 其中 `s`

`umOrOne fl` {- 每个子节点的叶子数量-} = 情况求和 `fl` 从 0 -> 1 -- 我的子节点没有叶子意味着我就是叶子 1 -> 1 -- 否则，传递总数

一个快速的例子，基于 Hutton 剃刀（一个包含整数和加法的表达式语言，它通常是用来说明这一点的最简单例子）。这些表达式是由 Hutton 的函子生成的。

数据 `HF h = Val Int | h :+: h` 派生 (Functor, Foldable, Traversable)

我介绍了一些模式同义词，以恢复定制字体的外观和感觉。

模式 `V x = 在 (Val x)` 模式 `s :+: t = 在 (s :+: t)`

我随手构造了一个简短的示例表达式，其中包含一些深度为三层的叶节点。

示例 :: 修复 HF 示例 = `(V 1 :+: V 2) :+: ((V 3 :+: V 4) :+: V 5)`

果然

好的，模块已加载：Leaves。\*Leaves

> leaves 示例 5

一种替代方法是在感兴趣的子结构中具有函子性和可折叠性，在这种情况下，就是叶子上的内容。（我们得到的正是自由单子。）

数据 树 `f x = 叶 x | 节点 (f (树 f x))` 派生 (函子, 可折叠)

一旦你将叶子/节点分离作为基本构建的一部分，你就可以直接使用 `foldMap` 访问叶子。加上一点 `Control.Newtype`，我们得到

`ala' Sum foldMap (const 1) :: Foldable f => f x -> Integer`

低于费尔贝恩阈值（即，足够短以至于不需要一个名字，并且因为没有名字而更为清晰）。

问题当然在于，数据结构通常在“感兴趣的子结构”中以多种有趣但相互冲突的方式具有函子性质。Haskell 并不总是最擅长让我们访问“已发现的函子性质”：我们必须在声明时预测我们需要的函子性质，当我们对数据类型进行参数化时。然而，仍然有时间改变这一切……

## 3.9 创建列表 评估列表元素

这是我对这个问题的思考过程……我们想要将一个列表分割成“链”（即一个列表的列表），给定一个测试来检查两个元素是否能够连接起来。

链 :: `(x -> x -> 布尔) -> [x] -> [[x]]`

我不记得库中有这样的东西，所以我决定自己实现一个。我想确定一种适合处理该列表的递归策略。

我可以只考虑元素吗？不：我很快排除了 `map` 和 `foldMap`，因为在这个问题中，元素似乎没有被独立处理。

接下来，我问“输出类型是否有 *list algebra*？”以这种方式表述，可能听起来不像是一个明显的问题，但它可以展开成以下合乎逻辑的问题。是否有构建输出（链表的列表）而不是输入（列表）的‘`nil`’和‘`cons`’操作？如果有，我可以使用 `foldr` 将输入的 `nil-and-cons` 转换为输出的 `nil-and-cons`，如下所示。

```
链 :: (x -> x -> 布尔值) -> [x] -> [[x]] 链接 = foldr chCons ch
Nil 其中 -- chNil :: [[x]] -- chCons :: x -> [[x]] -> [[x]]
```

显然，chNil 必须是什么，因为我正在对原始元素进行分组。输入为空？输出为空！

```
链 :: (x -> x -> 布尔) -> [x] -> [[x]] 链 链接 = foldr chCons []
其中 -- chCons :: x -> [[x]] -> [[x]]
```

我能写 chCons 吗？假设我得到一个链的列表：我该如何添加一个新元素？嗯，如果有一条可以链接的前端链，那么我就应该扩展那条链；否则我就应该开始一条新的链。因此，我为非空链列表的开头是一个非空链的情况设置了一个特殊分支，而默认情况则是用 cons 构造一个单元素。

```
链条 :: (x -> x -> 布尔) -> [x] -> [[x]] 链条 链接 = foldr chCons [] where chCons y (
xs@(x : _) : xss) | 链接 y x = (y : xs) : xss chCons y xss = [y] : xss
```

我们到家了！

```
> 链 (\ x y -> x + 1 == y) [1,2,3,4,5,6,8,9,10] [[1,2,3,4,5,6],[8,9,10]]
```

如果你能为某个类型的值实现这些算子，那么一组算子就对该给定类型拥有一个 *algebra*。数据类型构造器只是其中一种代数，是一组算子的一个实现，用来在该数据类型本身中构建值。使用来自某个数据类型的输入进行计算的一个好方法，是为你期望的输出类型实现它的代数。foldr 的要点在于捕捉这种“寻找代数”的模式，而它正好切中这个问题的要害。

### 3.10 GADT 的函数

你报告的错误并不是唯一的错误。

让我们戴上那副特殊的眼镜，它能显示那些通常被“类型推断”所隐藏的东西。

首先，数据构造器：

```
简单 :: forall a. (Typeable a, Show a) => 消息 -> (字符串 -> a) -
> 问题
```

实际上，一个 Question 类型的值看起来像是

```
舔狗 该 {a}{typeableDict4a}{showDict4a} 消息 解析器
```

在我把看不见的东西写在大括号里的地方，构造函数打包了一个类型和两个类型类字典，它们提供了 Typeable 和 Show 成员的实现。现在让我们来看主程序。我重命名了类型变量，以强调这一点。

```
runQuestion :: forall b. (Typeable b, Show b) => Question -> IO b
```

返回的类型由 runQuestion 的调用者选择，这与打包在 Question 类型参数中的任何类型是分开的。现在让我们在程序本身中填补那些不可见的组成部分。

```
runQuestion {b}{typeableDict4b}{showDict4b} (简单的 {a}{typeableDict4a}{showDict4a} 消息解析器) = do -- 因此 parser :: String -> a putStrLn message -- 可以, 因为 message :: String ans <- getLine -- 确保 ans :: String return $ parser ans -- 类型是 IO a, 而不是 IO b
```

解析器计算的是一个封装在 `Question` 中的 `a` 类型的值, 这与直接传递给 `runQuestion` 的 `b` 类型完全独立。该程序无法通过类型检查, 因为两个类型之间存在冲突, 而它们可以被程序的调用者设为不同。

与此同时, 让我们看看 `print`

```
打印 :: forall c. 显示 c => c -> IO ()
```

当你写

```
主函数 = 获取行 >>= (运行问题 . 获取问题) >>= 打印
```

你得到

```
主要 = getLine >>= runQuestion {b}{typeableDict4b}{showDict4b} . getQuestion) >>= 打印
( {b}{showDict4b}
```

作为 `runQuestion {b}` 的返回类型是 `IO b`, 因此 `print` 的 `c` 类型必须与 `runQuestion` 的 `b` 类型相同, 除此之外, 没有 *nothing* 来确定 `b` 的类型, 也没有理由它是 `Typeable` 或 `Show` 的实例。通过类型注解, 首先会出现对 `Typeable` 的需求 (在 `runQuestion` 调用中); 如果没有注解, 则 `print` 中对 `Show` 的需求会导致错误。

真正的问题在于, 不知为何, 你似乎希望 `runQuestion` 能返回一个其类型取决于问题中所隐藏类型的值, 仿佛你能够写出一个 (依赖类型的) 程序来做到这一点。

```
typeFrom :: 问题 -> * typeFrom (Simple {a}{typeableDict4a}{showDict4a} 消息解析器) = a
```

```
runQuestion :: (q :: Question) -> IO (typeFrom q)
```

这完全是一个合理的需求, 但这不是 `Haskell`: 没有办法给“封装在那个参数内部的类型”命名。所有涉及该类型的东西都必须处在将其暴露出来的 `case` 分析或模式匹配的作用域内。你试图在该作用域之外进行打印的做法是不被允许的。



## 第四章

# 应用函子

### 4.1 在哪里可以找到应用函子的编程练习？

把一些问题当作答案来发布似乎很有意思。这是一个有趣的例子，基于数独，探讨 `Applicative` 与 `Traversable` 之间的相互作用。

(1) 考虑

数据 三元组 `a = Tr a a a`

构建

实例 `Applicative Triple` 实例 `Traversable Triple`

以便 `Applicative` 实例执行“向量化”，并且 `Traversable` 实例按从左到右的顺序工作。别忘了构造一个合适的 `Functor` 实例：检查你是否可以从 `Applicative` 或 `Traversable` 实例中提取它。你可能会发现

新类型 `I x = I {unI :: x}`

对后者有用。

(2) 考虑

`newtype (..) f g x = Comp {comp :: f (g x)}`

证明

实例 `(Applicative f, Applicative g) => Applicative (f :. g)` 实例 `Traversable f, Traversable g`  
`=> Traversable (f :. g)`

现在定义

类型 区域 = 三重 `.. 三重`

假设我们将一个棋盘表示为一个垂直区域，由水平区域组成。

类型 板块 = 区域 `.. 区域`

展示如何利用 `traverse` 的功能，将其重新排列为由纵向区域组成的横向区域，以及由方块组成的正方形。

(3) 考虑

新类型 `Parse x = 解析器 {解析 :: 字符串 -> [(x, 字符串)]}` 派生 幺半群

或者其他合适的构造（注意到库中针对 `|Maybe|` 的 `Monoid` 行为并不合适）。构造

实例 `Applicative` 解析 实例 `Alternative` 解析 -- 只需遵循 ‘`Monoid`’

并实现

`ch :: (字符 -> 布尔值) -> 解析 字符`

如果被给定的谓词接受，它会消耗并输出一个字符。

(4) 实现一个解析器，该解析器消耗任意数量的空白字符，后跟一个单一数字（0 表示空格）

`平方 :: 解析 整数`

使用 `pure` 和 `traverse` 来构造

`板 :: 解析 (板 整数)`

(5) 考虑常值函子

新类型 `K a x = K {unK :: a}`

并构建

`instance Monoid a => Applicative (K a)`

然后使用 `traverse` 来实现

`crush :: (Traversable f, Monoid b) => (a -> b) -> f a -> b`

为 `Bool` 构造 `newtype` 包装器，以表达其合取和析取的幺半群结构。使用 `crush` 实现适用于任意 `Traversable` 函子的 `any` 和 `all` 的版本。

(6) 实现

`重复元素 :: (Traversable f, Eq a) => f a -> [a]`

计算出现多次的值列表。（并非完全简单。）（有一种很棒的方法可以使用微积分来完成，但那是另一个话题。）

(7) 实现

`完成 :: Board Int -> Bool` 好 `:: Board Int -> Bool`

检查一个棋盘是否满足以下条件：(1) 仅包含 `[1..9]` 范围内的数字，(2) 任一行、列或盒子内没有重复数字。

## 4.2 N元树遍历

### 4.2.1 第一次尝试：艰苦的工作

对于n元树的情况，有*three*件事情发生：元素编号、树的编号，以及*lists*棵树的编号。将它们分开处理会有所帮助。首先是类型：

```
aNumber :: a -- 要编号的对象 -> Int -- 起始编号 -> ( (a, Int) -- 已编号的对象, Int -- 之后
可用的下一个编号 )
ntNumber :: NT a -- 要编号的对象 -> Int -- 起始编号 -> ( NT (a, Int) -
- 已编号的对象, Int -- 之后可用的下一个编号 )
ntsNumber :: [NT a] -- 要编号的对象 -> Int
-- 起始编号 -> ( [NT (a, Int)] -- 已编号的对象, Int -- 之后可用的下一个编号 )
```

注意到这三种类型都共享同一种模式。当你发现自己似乎是偶然地在遵循某个模式时，你就知道你有机会学到一些东西。不过我们先继续往前，稍后再学习。

给一个元素编号很容易：将起始编号复制到输出中，并将它的后继作为下一个可用的编号返回。

```
aNumber a i = ((a, i), i + 1)
```

对于另外两个，模式（又是这个词）是

1. 将输入分割成其顶级组件 2. 依次为每个组件编号，并将数字串联起来

用模式匹配（对数据进行目视检查）来完成第一项很容易，而用 `where` 子句（获取输出的两个部分）来完成第二项也很容易。

对于树结构，顶层分裂给我们两个组成部分：一个元素和一个列表。在`where`子句中，我们根据这些类型调用适当的编号函数。在每种情况下，“thing”输出告诉我们在“thing”输入的位置放置什么。同时，我们将数字贯穿其中，因此整个结构的起始数字就是第一个组成部分的起始数字，第一个组成部分的“下一个”数字开始第二个，第二个的“下一个”数字就是整个结构的“下一个”数字。

```
ntNumber (N a ants) i0 = (N ai aints, i2) 其中 (ai, i1) = aNumber a i0
(aints, i2) = ntsNumber ants i1
```

对于列表，我们有两种可能性。一个空列表没有组件，因此我们直接返回它，而不使用更多的数字。一个“cons”有两个组件，我们按照之前的做法，使用由类型指示的适当编号函数。

```
ntsNumber [] i = ([], i) ntsNumber (ant : ants) i0 = (aint : aints, i2) where
(aint, i1) = ntNumber ant i0 (aints, i2) = ntsNumber ants i1
```

我们试试看。

```
> let ntree = N "eric" [N "lea" [N "kristy" [], N "pedro" [], N "rafael" []], N "anna" []]
> ntNumber ntree 0 (N ("eric",0) [N ("lea",1) [N ("kristy",2) [], N ("pedro",3) [], N ("rafael",4) []], N ("a
```

所以我们算是到这一步了。但我们开心吗？嗯，我不开心。我有一种恼人的感觉：我几乎把同一种类型写了三遍，又把几乎同一个程序写了两遍。而且如果我想针对组织方式不同的数据（例如你的二叉树）做更多的元素编号，我就不得不一遍又一遍地写同样的东西。Haskell 代码中的重复模式是 *always* 被错过的机会：培养自我批判的意识，并问一问是否有更简洁的做法，是很重要的。

#### 4.2.2 第二次尝试：编号和螺纹

我们看到的两个重复模式是：1. 类型的相似性，2. 数字穿插方式的相似性。

如果你匹配把这些类型放在一起看看有什么共同之处，你会注意到 你们都是  
输入 -> 整数 -> ( 输出, 整数)

对于不同的输入和输出。我们给最大的公共组件命名。

类型编号输出 = 整数 -> ( 输出, 我 nt)

现在我们的三种类型是

```
aNumber :: a -> 编号 (a, Int) ntNumber :: NT a -> 编号 (NT (a, Int))
ntsNumber :: [NT a] -> 编号 [NT (a, Int)]
```

你经常在 Haskell 中看到这种类型：

输入 -> 执行操作以获取输出

现在，为了处理线程问题，我们可以构建一些有用的工具来进行操作并结合编号操作。为了查看我们需要哪些工具，看看我们在对组件进行编号后是如何结合输出的。输出中的“事物”部分总是通过将一些未编号的函数（通常是数据构造器）应用于来自编号的“事物”输出来构建的。

为了处理这些函数，我们可以构建一个与我们的[]案例非常相似的工具，其中不需要实际的编号。

```
稳定 :: 事物 -> 编号 事物 稳定 x i = (x, i)
```

不要被排版方式迷惑，以为 `steady` 只有一个参数：要记住，`Numbering` 这个东西是对函数类型的缩写，所以里面实际上还有另一个 `->`。于是我们得到

```
稳定 [] :: 编号 [a] 稳定 [] i = ([], i)
```

就像在`ntsNumber`的第一行一样。但是其他构造函数，`N`和`(:)`呢？问问`ghci`。

```
> :t 稳定 N 稳定 N :: 编号 (a -> [NT a] -> NT a) > :t 稳定 (:)
稳定 (:) :: 编号 (a -> [a] -> [a])
```

我们通过 *functions* 获取编号操作作为输出，并希望通过更多的编号操作生成这些函数的参数，从而产生一个包含这些数字的大型总体编号操作。这个过程的一个步骤是将一个编号生成的函数与一个编号生成的输入相结合。我将其定义为一个中缀运算符。

```
($$) :: 编号 (a -> b) -> 编号 a -> 编号 b infixl 2 $$
```

与显式应用算符的类型比较，`$`

```
> :t ($) ($) :: (a -> b) -> a -> b
```

该`$$`运算符是“编号应用”。如果我们能够正确使用它，我们的代码将变得

```
ntNumber :: NT a -> 编号 (NT (a, Int)) ntNumber (N a ants) i = (稳定 N $$ aNumber a $$ ntsNumber
ants) i
```

```
ntsNumber :: [NT a] -> 编号 [NT (a, Int)] ntsNumber [] i = 稳定 [] i ntsNumber (ant : ants) i = (稳定 (:) $$ n
tNumber ant $$ ntsNumber ants) i
```

随着 `aNumber` 按照当前的状态进行（暂时）。这段代码只是进行数据重建，将构造器和组件的编号过程拼接在一起。我们最好先给出 `$$` 的定义，并确保它正确地进行线程处理。

```
($$) :: 编号 (a -> b) -> 编号 a -> 编号 b (fn $$ an) i0 = (f a, i2) 其中 (f, i1) = fn
i0 (a, i2) = an i1
```

在这里，我们的旧线程 *pattern* 完成了 *once*。`fn` 和 `an` 都是函数，期望一个起始数字，而整个 `f n $$ sn` 是一个函数，它获取起始数字 `i0`。我们通过线程传递这些数字，首先收集函数，然后是参数。接着我们进行实际的应用，并返回最终的“下一个”数字。

现在，请注意，在每一行代码中，`i` 输入作为编号过程的参数传入。我们可以通过仅讨论 *processes* 来简化这段代码，而不是 *numbers*。

```
ntNumber :: NT a -> 编号 (NT (a, Int)) ntNumber (N a ants) = 稳定的 N $$ aNumber a $$ nts
Number ants
```

```
ntsNumber :: [NT a] -> 编号 [NT (a, Int)] ntsNumber [] = 稳定 [] ntsNumber (ant : ants) = 稳定 (:) $$
ntNumber ant $$ ntsNumber ants
```

一种读取此代码的方法是过滤掉所有的编号、稳定和`$$`用途。

```
ntNumber :: NT a -> ..... (NT (a, Int)) ntNumber (N a ants) = ..... N .. (aNumber a) .. (ntsNumber a
nts)
```

```
ntsNumber :: [NT a] -> ..... [NT (a, Int)] ntsNumber [] = ..... [] ntsNumber (ant : ants) = ..... (:) .. (ntNumb
er ant) .. (ntsNumber ants)
```

而你会发现它看起来就像是一个前序遍历，在处理元素后重建原始数据结构。我们正在正确地使用 *values*，前提是稳态和 \$\$ 正确地结合了 *processes*。

我们可以尝试对 *aNumber* 做同样的事情

```
aNumber :: a -> 编号 a
一个数字 a = 稳定 (,) $$ 稳定 a $$ ????
```

但是 ??? 才是我们实际上需要这个数字的地方。我们可以构建一个适合填补那个空缺的编号过程：一个 *issues the next number* 的编号过程。

```
下一个 :: 编号 整数 下一个 i
= (i, i + 1)
```

这就是编号的本质，“thing” 输出的是现在要使用的数字（即起始数字），而“next” 输出的是下一个数字。我们可以写作

```
一个数字 a = 稳定 (,) $$ 稳定 a $$ 下一个
```

可简化为

```
一个数字 一个 = 稳定 ((,) 一个) $$ 下一步
```

在我们经过筛选的视角中，那就是

```
一个数字 a = ..... ((,) a) .. 下一个
```

我们所做的是将“编号过程”的概念进行了封装，并且我们构建了合适的工具来用这些过程执行 *ordinary functional programming*。线程模式转化为稳态和 \$\$ 的定义。

Nu 记忆并不是唯一以这种方式运作的东西。试试这个 s...

```
> :info 应用
类 Functor f => Applicative (f :: * -> *) 其中 pure :: a -> f a (<*>) :: f (a -
> b) -> f a -> f b
```

……你还会得到一些额外的东西。我只是想提醒一下 *pure* 和 *<\*>* 的类型。它们很像 *steady* 和 *\$\$*，但不仅仅用于编号。*Applicative* 是用于 *every* 这类以这种方式运作的过程的类型类。我不是在说“现在就去学 *Applicative*！”，只是建议一个前进的方向。

### 4.2.3 第三次尝试：类型引导编号

到目前为止，我们的解决方案针对的是一种特定的数据结构 *NT a*，而 *[NT a]* 作为一个辅助概念出现，因为它被用于 *NT a* 中。我们可以通过一次只关注类型的一层，使整体更加即插即用。我们通过对树进行编号来定义对树列表的编号。一般来说，如果我们知道如何为 *stuff* 的每个元素编号，就知道如何为 *stuff* 的列表编号。

如果我们知道如何对一个 *a* 编号以得到 *b*，那么我们应该能够对一个 *a* 的 *list* 编号，以得到一个 *b* 的 *list*。我们可以对“如何处理每个项目”进行抽象。

```
listNumber :: (a -> Numbering b) -> [a] -> Numbering [b]
listNumber na [] = steady [] listNumber
mber na (a : as) = steady (:) $$ na a $$ listNumber na as
```

现在我们的旧树木编号函数变成了

```
ntsNumber :: [NT a] -> 编号 [NT (a, Int)]
ntsNumber = listNumber
ntNumber
```

几乎不值得一提。我们可以直接写

```
ntNumber :: NT a -> 编号 (NT (a, Int))
ntNumber (N a ants) = 稳定 N $$ aNumber a $$ listNumber ntNumber ants
```

我们可以对树本身进行相同的游戏。如果你知道如何编号物品，你就知道如何为一棵物品树编号。

```
ntNumber' :: (a -> 编号 b) -> NT a -> 编号 (NT b)
ntNumber' na (N a 蚂蚁) = 稳定 N $$ na a $$ listNumber (ntNumber' na) 蚂蚁
```

现在我们可以做这样的事情

```
myTree :: NT [String]
myTree = N ["a", "b", "c"] [N ["d", "e"] [], N ["f"] []]
```

```
> ntNumber' (listNumber aNumber) myTree 0 (N [ ("a", 0) , ("b", 1) , ("c", 2) ] [N [ ("d", 3) , ("e", 4) ] [], N [ ("f", 5) ] []], 6)
```

这里，节点数据现在本身是一个事物的列表，但我们已经能够单独编号这些事物。我们的设备更加灵活，因为每个组件都与类型的一个层次对齐。

现在，试试这个：

```
> :t 遍历
遍历 :: (应用性 f, 可遍历 t) => (a -> f b) -> t a -> f (t b)
```

这非常像我们刚才做的事情，其中 `f` 是编号，`t` 有时是列表，有时是树。

`Traversable` 类刻画了作为一种类型构造器的含义：它允许你将某种过程贯穿于所存储的元素之中。同样，你正在使用的这种模式非常常见，早已被预先考虑到。学会使用 `traverse` 能节省大量工作。

#### 4.2.4 最终。。。。

……你将会了解到，用来完成编号工作的东西在库中已经存在：它叫做 `State Int`，属于 `Monad` 类，这意味着它也一定属于 `Applicative` 类。要获取它，

导入 `Control.Monad.State`

而那个用其初始状态启动一个有状态过程的操作，比如我们输入 0 的那个，就是这个：

```
> :t evalState
evalState :: State s a -> s -> a
```

我们的下一个操作是

```
next' :: State Int Int next' = 获取 <
* 修改 (1+)
```

其中 `get` 是访问状态的过程，`modify` 是改变状态的过程，而 `<*` 意味着“但也要做”。

如果你在文件开头使用语言扩展指令

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}
```

你可以像这样声明你的数据类型

```
数据 NT a = N a [NT a] 导出 (Show, Functor, Foldable, Traversable)
```

Haskell 会为你编写 `traverse`。然后你的程序变成一行。

```
evalState (traverse (\ a -> pure ((,) a) <*> get <*> modify (1+)) ntree) 0 -- 如何处理一个元素
~~~~~-- 如何处理整个树形元素 ~~~~~-- 处理你的特定树形
~~~~~-- 从起始数字0开始启动处理过程 ~~~~~
```

……但通向那一行的旅程涉及许多“封装模式”的步骤，这需要一些（希望是有回报的）学习。

### 4.3 函数的偏应用与柯里化：如何编写更好的代码，而不是大量使用 `map`?

到目前为止所有的建议都很好。这是另一个建议，起初可能看起来有点奇怪，但在许多其他情况下非常有用。

一些类型构造算子，比如 `[]`，它是将元素的类型（例如 `Int`）映射到由这些元素组成的列表类型 `[Int]` 的算子，具有 `Applicative` 的性质。对于列表而言，这意味着存在某种方式，由算子 `<*>`（读作“`apply`”）表示，用来把 `lists` 的函数和 `lists` 的参数转换成 `lists` 的结果。

```
(<*>) :: [s -> t] -> [s] -> [t] -- <*>的一般类型的一个实例
```

而不是由空白空间或 `$` 给出的普通应用，

Please provide the text you would like to have translated.

结果是，我们可以使用事物的列表而不是事物本身来进行普通的函数式编程：我们有时称之为“在列表 *idiom* 中编程”。唯一的其他要素是，为了应对某些组件是单独的事物的情况，我们需要一个额外的工具。

纯 `:: x -> [x]` -- 再次，通用方案的一个实例

它将某个东西包装成一个列表，以便与 `<*>` 兼容。这就是 `pure`：把一个普通值移入 `applicativ` e 习语中。

对于列表，`pure` 只是生成一个单例列表，`<*>` 产生每一对函数与一个参数的配对应用结果。特别是

```
纯 f <*> [1..10] :: [Int -> Int -> Int -> Int -> Int]
```

是一个函数列表（就像 `map f [1..10]`），可以再次与 `<*>` 一起使用。你传给 `f` 的其余参数都不是列表式的，所以你需要用 `pure` 把它们包装起来。

```
纯 f <*> [1..10] <*> 纯 1 <*> 纯 2 <*> 纯 3 <*> 纯 4
```

对于列表，这将得到

```
[f <*> [1..10] <*> [1] <*> [2] <*> [3] <*> [4]]
```

即：从 `f` 构造应用的方法列表，`[1..10]` 中的一个：1、2、3 和 4。

开局的纯 `f <*> s` 非常常见，因此被缩写为 `f <$> s`，所以

```
f <$> [1..10] <*> [1] <*> [2] <*> [3] <*> [4]
```

通常会写成这样。如果你能过滤掉 `<$>`、纯粹的和 `<*>` 噪音，它看起来就像你想要的应用程序。额外的标点符号只是因为 `Haskell` 无法区分一系列函数或参数的列表计算和意图作为单一值但恰好是列表的非列表计算。不过，至少组件的顺序是你开始时设定的，因此你更容易看出发生了什么。

玄学。（1）在我（不是很）私人化的 `Haskell` 方言中，上面的代码将是

```
(f [1..10] (|1|) (|2|) (|3|) (|4|) |)
```

其中每个 *idiom bracket*、`(f a1 a2 ... an)` 都表示将一个纯函数应用于零个或多个存在于 *idiom* 中的参数。这只是一中书写方式

```
纯 f <*> a1 <*> a2 ... <*> an
```

`Idris` 有习语括号，但 `Haskell` 还没有添加它们。还没有。

在具有代数效应的语言中，非确定性计算的习语（对于类型检查器来说）与列表的数据类型不同，尽管你可以很容易地在两者之间转换。程序变成

```
f (范围 1 10) 2 3 4
```

其中 `range` 以非确定性的方式在给定的下界和上界之间选择一个值。因此，非确定性被视为一种 *local* 副作用，而不是一种数据结构，从而支持失败和选择等操作。你可以将非确定性的计算包装在一个 *handler* 中，它为这些操作赋予语义，其中一种这样的处理器可能会生成所有解的列表。也就是说，用来解释正在发生什么的额外记法被推到边界上，而不是像那些 `<*>` 和 `pure` 那样散布在整个内部。

管理事物的边界而非其内部，是我们这个物种少数几个成功想出的好主意之一。但至少这个主意可以被我们一再地使用。这就是为什么我们选择耕作而不是狩猎。这也是为什么我们更偏好静态类型检查而不是动态标签检查。诸如此类……

## 4.4 将单子转换为应用范畴

我会写

```
integer :: Parser Integer
integer = read <$ many1 space <*> many1 digit
```

有一堆左结合的（如应用）解析器构建操作符  $\langle \$ \rangle$ 、 $\langle * \rangle$ 、 $\langle \$ \rangle$ 、 $\langle * \rangle$ 。最左边的部分应该是纯函数，它将从组件值中组装结果值。每个操作符右边的部分应该是解析器，整体上按从左到右的顺序给出语法的组件。选择使用哪个操作符取决于以下两个选择。

```

the thing to the right is      signal / noise
-----
the thing to the left is \
                        +-----+
                        pure / |  <$>      <$
                        a parser |  <*>      <*>

```

所以，在选择 `read :: String -> Integer` 作为将要提供解析器语义的纯函数之后，我们可以将前导空格归类为“噪声”，而那一串数字归类为“信号”，因此

```

read <$ many1 space <*> many1 digit
(..)      (.....)      (.....)
pure      noise parser   |
(.....)      |
      parser           signal parser
(.....)
                        parser

```

你可以结合多种可能性与

```
p1 <|> ... <|> pn
```

并用  $\{v^*\}$  表示不可能性

It looks like you didn't provide any source text for translation. Could you please share the text you'd like to have translated?

在解析器中很少需要命名组件，生成的代码更像是一个带有附加语义的语法。

## 4.5 应用词汇可组合，单子不可组合

如果我们比较这些类型

```

(<*>) :: 应用函子 a => a (s -> t) -> a s -> a t (>=>) :: 单子 m => m s -
> (s -> m t) -> m t

```

我们得到了区分这两个概念的线索。 $(>=>)$  的类型中出现的  $(s -> m t)$  表明， $s$  中的一个值可以决定  $m t$  中一次计算的行为。单子允许值层与计算层之间的相互干扰。 $\langle * \rangle$  运算符不允许这种干扰：函数和参数的计算不依赖于值。这一点影响非常大。比较

```

miffy :: 单子 m => m 布尔值 -> m x -> m x -> m x miffy mb mt mf
= do b <- mb 如果 b 那么 mt 否则 mf

```

它使用某些效果的结果来决定在两个 *computations* (例如，发射导弹和签署停战协议)之间。

iffy :: Applicative a => a 布尔 -> a x -> a x -> a x  
 iffy ab at af = pure cond <  
 \*> ab <\*> at <\*> af 其中 cond b t f = 如果 b 那么 t 否则 f

它使用 ab 的值在 *the values of* 两种计算 at 和 af 之间进行选择，却同时执行了两者，或许造成了悲剧性的后果。

单态版本本质上依赖于 (>>=) 的额外能力来从一个值中选择计算，这一点可能很重要。然而，支持这种能力使得单子难以组合。如果我们尝试构建“双重绑定”

```
(>>>==) :: (Monad m, Monad n) => m (n s) -> (s -> m (n t)) -> m (n t)
mns >>>== f = mns >>-{-m-}
\ ns -> 令 nmnt = ns >>= (return . f) 在 ???
```

我们做到这一点，但现在我们的层次都混乱了。我们有一个  $n(m(n t))$ ，所以我们需要去掉外层的  $n$ 。正如 Alexandre C 所说，如果我们有一个合适的，我们就能做到。

交换 ::  $n(m t) \rightarrow m(n t)$

将  $n$  向内置换，并将其连接到另一个  $n$ 。  
 较弱的“重复应用”更容易定义

```
(<<*>>) :: (Applicative a, Applicative b) => a (b (s -> t)) -> a (b s) -> a (b t)
abf <<*>> abs = pure (<*>) <*> abf <*>
abs
```

因为层与层之间没有干扰。

相应地，认识到何时真正需要 Monad 的额外能力，以及何时可以依赖 Applicative 支持的严格计算结构，是很重要的。

请注意，虽然组合单子很困难，但它可能超出了你的需求。类型  $m(n v)$  表示先计算  $m$ -效果，再计算  $n$ -效果得到  $v$ -值，其中  $m$ -效果在  $n$ -效果开始之前完成（因此需要 swap）。如果你只想将  $m$ -效果和  $n$ -效果交替执行，那么组合可能就不必要了！

## 4.6 区分 Functor、Applicative 和 Monad 的示例

我的风格可能受到手机的限制，但我试试看。

```
newtype Not x = Kill {kill :: x -> Void}
```

不能是一个函子。如果是的话，我们就会有

```
kill (fmap (const ()) (Kill id)) () :: Void
```

而月亮会被做成绿奶酪。与此同时

```
新类型 Dead x = 哎呀 {oops :: Void}
```

是一个函子

实例 Functor Dead, 其中 fmap f (Oops corpse) = O  
 ops corpse

但不能是应用式的，否则我们就会有

哎呀 (纯粹 ()) :: 空

而绿色会由月亮奶酪制成（这实际上确实可能发生，但只会在晚上较晚的时候）。

（附注：Void，如同 Data.Void 中的含义，是一个空数据类型。如果你试图用 undefined 来证明它是一个 Monoid，我会用 unsafeCoerce 来证明它不是。）

愉快地，

```
newtype Boo x = Boo {boo :: Bool}
```

在许多方面具有应用性，例如，用狄克斯特拉的话来说，

实例 Applicative Boo 其中 pure \_ = Boo True Boo b

```
1 <*> Boo b2 = Boo (b1 == b2)
```

但它不能是一个单子。要了解为什么不行，请注意，return 必须始终是 Boo True 或 Boo False，因此

加入 . 返回 == id

根本不可能成立。哦，  
对了，我差点忘了

新类型 Thud x = 该 {only :: ()}

是一个 Monad。自己动手做  
。要赶飞机……

## 4.7 Parsec：Applicative 与 Monad 的对比

为了判断各自何时适用，关注 Applicative 和 Monad 之间的关键语义差异可能是值得的。比较类型：

```
(<*>) :: m (s -> t) -> m s -> m t
(>>=) :: m s -> (s -> m t) -> m t
```

要部署<\*>，你选择两个计算：一个是函数的计算，另一个是参数的计算，然后通过应用将它们值组合起来。要部署>>=，你选择一个计算，并说明你将如何利用其产生的值来选择下一个计算。这就是“批处理模式”和“交互式”操作之间的区别。

在解析方面，Applicative（通过加入失败和选择扩展为 Alternative）刻画了你的语法的 *context-free* 个方面。只有当你需要检查来自输入某一部分的解析树，以决定在输入的另一部分应使用什么语法时，才需要 Monad 所提供的额外能力。例如，你可能先读取一个格式描述符，然后再读取符合该格式的输入。尽量减少对 monad 额外能力的使用，可以告诉你哪些值依赖是必不可少的。

从解析转向并行性，这种只在必要的值依赖上使用 >>= 的想法，让你清楚地看到分摊负载的机会。当两个计算通过 <\*> 组合时，彼此都无需等待对方。能用 Applicative 就用，用到必须时才用 Monadic，这是速度的公式。ApplicativeDo 的要点在于，对以单子风格编写、从而不小心被过度顺序化的代码，自动化其依赖分析。

你的问题也涉及编码风格，而对此的看法本就见仁见智。不过让我给你讲个故事。我从 Standard ML 转到 Haskell；在 ML 中，我习惯于用直接风格来写程序，即便它们会做一些“顽皮”的事情，比如抛出异常或修改引用。我在 ML 里做什么呢？在参与一个极端纯粹的类型理论的实现（出于法律原因，不能点名）。在处理 *in* 那套类型理论时，我无法使用直接风格来编写。

使用异常的程序，但我构思出了应用式组合子，作为尽可能接近直接风格的一种方式。

当我转向 Haskell 时，我震惊地发现，人们似乎普遍认为，用伪命令式的 `do` 记法进行编程，只是对哪怕最轻微的语义不纯的一种惩罚（当然，非终止性除外）。早在我真正理解其中的语义差异之前，我就已经把应用式组合子当作一种风格选择来采用了（并且借助“习语括号”进一步接近直接风格），也就是说，在我明白它们代表的是对 `monad` 接口的一种有用的弱化之前。我只是（过去如此，现在仍然如此）不喜欢 `do` 记法要求打碎表达式结构，以及对事物进行无端命名的方式。

也就是说，使函数式代码比命令式代码更紧凑、更可读的那些因素，同样也使得应用式风格比 `do-notation` 更紧凑、更可读。我承认，`ApplicativeDo` 是一种很好的方式，可以把那些以单子风格编写、而你又没有时间重构的程序，改写成更应用式的（在某些情况下这意味着 *faster*）。但除此之外，我认为“能用应用式就用应用式，非用单子不可时再用单子”也是理解正在发生什么的更好方式。

## 4.8 将do表示法重构为应用式风格

所有这些运算符都是左结合的；`<` 和/或 `>` 指向那些贡献值的事物；其中，`$` 表示左侧是纯值，`*` 表示左侧是应用式计算。

我使用这些运算符的经验法则如下。首先，列出你的语法生成的各个组成部分，并根据它们是否在语义上提供重要信息，将其分类为“信号”或“噪声”。在这里，我们有

```
字符 '(' -- 噪声 buildExpr --
信号 字符 ')' -- 噪声
```

接下来，弄清楚什么是“语义函数”，它接受信号组件的值并给出整个产出的值。在这里，我们有

```
id —— 纯语义函数，然后是一堆组件解析器 char '(' —— 噪声 buildExpr —— 信号 char ')'
—— 噪声
```

现在，每个组件解析器需要与之前的部分通过操作符连接，但使用哪个操作符呢？

- 始终从 `<` 开始
- 下一个 `$` 对于第一个组件（如纯函数之前的那样），或者 `*` 对于每个其他组件
- 然后是 `>`，如果组件是 *signal* 或如果它是 *noise*

因此我们得到

```
id -- 纯语义功能，然后是一堆解析器 <$ char '(' -- 首先，噪声 <*> buildExpr
-- 后来，信号 <*> char ')' -- 后来，噪声
```

如果语义函数是 `id`（如这里所示），你可以把它去掉，直接使用 `>` 将噪声粘接到信号前面，而该信号正是 `id` 的参数。我通常选择不这么做，只是为了能清楚地看到语义函数明确地位于产生式的开头。此外，你还可以通过穿插 `<|>` 在这些产生式之间构建一个选择，而且不需要用括号把它们包起来。

## 4.9 使用默认值进行压缩，而不是丢弃值？

有一些 `st` 为这个问题建立结构，它来了。我将会使用 `th` 是东西：

```
导入 Control.Applicative 导入 Data.T
raversable 导入 Data.List
```

首先，带填充的列表是一个有用的概念，因此我们为它们定义一个类型。

数据 `Padme m = (:-) { padded :: [m], padder :: m }` 派生 (`Show`, `Eq`)

接下来，我记得截断-`zip`操作会产生一个 `Applicative` 实例，在库中是新类型 `ZipList`（一个流行的非 `Monad` 示例）。 `Applicative ZipList` 相当于通过无穷大和最小值装饰给定的单一元素。 `Padme` 具有类似的结构，只是它的底层单一元素是正数（带有无穷大），使用一和最大值。

```
我instance Applicative Padme where pure = ([] :-) fs :- f) <*> (ss :- s
) = zapp fs ss :- f s where zapp [] ss = map f ss zapp fs [] = map ($ s) fs z
app (f : fs) (s : ss) = f s : zapp fs ss
```

我有责任念出通常的咒语以生成默认的 `Functor` 实例。

```
instance Functor Padme where fmap = (<*>) . pure
```

因此，配备了这些工具，我们可以开始填充了！例如，那个将一个不规则的字符串列表填充空格的函数，变成了一行代码。

```
deggar :: [String] -> [String] deggar = transpose . padded . traverse (-)
)
```

看到了吗？

```
*Padme> deggar ["om", "mane", "padme", "hum"] ["om ", "mane ",
"padme", "hum "]
```

## 4.10 在 Haskell 中使用 `zipWith3` 进行 `sum3`

我以前见过这种问题，见于这里：<https://stackoverflow.com/q/21349408/828361> 我对那个问题的回答也适用于这里。

带有指定填充元素的 `ZipList` 应用函子列表是应用函子（该应用函子由正整数上的 1 与 `max` 的幺半群结构构建而成）。

数据 `Padme m = (:-) { padded :: [m], padder :: m }` 衍生 (`Show`, `Eq`)

```
我实例应用性 Padme, 其中 pure = ([] :-) fs :- f) <*> (ss :- s) = zapp
fs ss :- f s 其中 zapp [] ss = map f ss zapp fs [] = map ($ s) fs zapp (f : fs) (
s : ss) = f s : zapp fs ss
```

```
-- 和对于那些没有 DefaultSuperclassInstances 实例 Functor Padme 的你们来说, fmap = (<
*>) . pure
```

现在我们可以将数字列表连同其适当的填充一起打包起来

```
pad0 :: [Int] -> Padme Int
pad0 = (:-0)
```

这就给出了

填充后的  $((\lambda x y z \rightarrow x+y+z) \langle \$ \rangle \text{pad0 } [1,2,3] \langle * \rangle \text{pad0 } [4,5] \langle * \rangle \text{pad0 } [6]) = [11,7,3]$

或者，在成语括号不可用的情况下，你可以这样写

填充的  $(\mid \text{pad0 } [1,2,3] + (\mid \text{pad0 } [4,5] + \text{pad0 } 6 \mid) \mid)$

意思相同。

Applicative 为您提供了一种很好的方式来封装这个问题所要求的“填充”这一核心思想。

## 4.11 ‘Const’ 应用函子有什么用？

它与可遍历 (Traversable) 结合时非常有用。

```
getConst . t      raverse Const :: (Monoid a, Traversable f)      => f a -> a
```

这就是把一堆东西拼在一起的通用做法。这是让我确信将 Applicative 与 Monad 分离是值得的使用场景之一。我需要像 generalized elem 这样的东西

```
elem :: Eq x => x -> Term x -> Bool
```

为了对一个以自由变量表示为参数的可遍历 Term 进行出现检查。我不断地更改 Term 的表示方式，已经厌倦了修改成百上千个遍历函数，其中一些是在做累积，而不是进行有副作用的映射。很高兴找到了一个同时涵盖这两者的抽象。

## 4.12 自由单子的应用实例

这样可以吗？

实例 (函子 f) => 应用函子 (自由 f) 其中 pure = 返回 返回 f <\*> 作为 = 映射 f 作为 卷 faf <\*> 作为 = 卷 (映射 (<\*> 作为) faf)

计划是只在生成函数的树的叶节点上进行操作，因此对于 Return，我们通过将该函数应用到由参数动作产生的所有参数值来进行操作。对于 Roll，我们只是把打算对整体动作所做的事情同样施加到所有子动作上。

关键在于，当我们到达 Return 时要做什么，在开始之前就已经确定了。我们不会根据自己在树中的位置来改变计划。这正是 Applicative 的标志：计算的结构是固定的，因此值依赖于值，而动作不依赖于值。

### 4.13 可以比 Monad 部分更好地优化的 Applicative 部分的单子示例

也许最典型的例子是由这些向量给出的。

数据 Nat = Z | S Nat 派生 (Show, Eq, Ord)

数据 Vec :: Nat -> \* -> \* 其中 V0 :: Vec Z x (:>) :: x ->

Vec n x -> Vec (S n) x

我们只需稍加努力就能让它们成为 applicative，先定义单例，然后将它们包装进一个类中。

数据 Natty :: Nat -> \* 其中 Zy :: Natty Z Sy :

: Natty n -> Natty (S n)

类 NATTY (n :: Nat) 其中 natty :: Natty  
n

实例 NATTY Z, 其中 natty =  
Zy

实例 NATTY n => NATTY (S n) 其中 natty = Sy natty

现在我们可以发展应用结构

实例 NATTY n => Applicative (Vec n), 其中 pure = vcopies natty  
y (<\*>) = vapp

vcopies :: 对于所有 n x. Natty n -> x -> Vec n x vcopies Zy x = V0

vcopies (Sy n) x = x :> vcopies n x

vapp :: forall n s t. Vec n (s -> t) -> Vec n s -> Vec n t vapp V0 V0 = V0 vapp (f :> fs)

(s :> ss) = f s :> vapp fs ss

我省略了 Functor 实例（它应该通过 fmapDefault 从 Traversable 实例中提取）。

现在，有一个对应于这个 Applicative 的 Monad 实例，但它是什么呢？*Diagonal thinking! That's what's required!* 向量可以看作是从有限域到函数的表格化，因此 Applicative 只是 K- 和 S-组合子的表格化，而 Monad 具有类似于 Reader 的行为。

vtail :: forall n x. Vec (S n) x -> Vec n x vtail (x :> xs) = xs

vjoin :: forall n x. Natty n -> Vec n (Vec n x) -> Vec n x vjoin Zy \_ = V0 vjoin (Sy n) ((x :> \_) :> xxss) = x :> vjoin n (fmap vtail xxss)

实例 `NATTT n => Monad (Vec n)` 其中 `返回 = vcopies`  
`natty xs >>= f = vjoin natty (fmap f xs)`

通过更直接地定义 `>>=`，你也许能省下一点，但无论怎么切分，单子的行为都会为非对角计算生成无用的 `thunk`。惰性求值或许能避免我们以末日般的因子变慢，但 `<*>` 的拉链式行为至少肯定会比取一个矩阵的对角线要便宜一些。

## 4.14 “ap” 在单子实现中的任意性有多大？

这个问题至少有三个相关方面。

1. 给定一个 `Monad m` 实例，它的必需 `Applicative m` 超类实例的规范是什么？*Answer*: `pure` 是 `return`，`<*>` 是 `ap`，因此

`mf <*> ms == 执行 f <- mf; s <- ms; 返回 (f s)`

请注意，这个规范并不是 `Applicative` 类型类的一条定律。它是对 `Monad` 的一项要求，用以确保一致的使用模式。

2. 给定候选实现的规范，`ap` 是唯一可接受的实现吗？*Answer*: 响亮的回答是，不。`>>=` 类型所允许的值依赖有时会导致低效的执行：在某些情况下，`<*>` 可以比 `ap` 更高效，因为你无需等到第一次计算完成才能知道第二次计算是什么。“应用式 `do`” 符号正是为了利用这一可能性而存在的。

3. 是否还有其他 `Applicative` 的候选实例，尽管它们与所要求的 `ap` 实例不一致，却仍然满足 `Applicative` 定律？*Answer*: 是的。题目提出的“反向 (backwards)”实例正是这样一种情况。事实上，正如另一则回答所指出的，任何 `applicative` 都可以被反转，而结果往往是一个不同的事物。

对于读者的进一步示例和练习，请注意，非空列表在结构上是单子的，方式与普通列表中的单子类似。

数据 `Nellist x = x :& 也许 (Nellist x)`

`necat :: Nellist x -> Nellist x -> Nellist x`  
`necat (x :& Nothing) ys = x :&`  
`Just ys`  
`necat (x :& Just xs) ys = x :& Just (necat xs ys)`

实例 `Monad Nellist` 其中 `return x = x :& Nothing`  
`(x :& Nothing) (x :& Nothing)`  
`>>= k = k x (x :& Just xs)`  
`>>= k = necat (k x) (xs >>= k)`

找到至少 *four* 个行为上不同的 `Applicative Nellist` 实例，这些实例遵循 `Applicative` 法则。

## 4.15 不需要 Functor 的 Applicative（针对数组）

阅读评论后，我有点担心这里忽略了大小问题。当尺寸不匹配时，是否有合理的行为？

同时，可能有一些方法你可以合理地沿着以下方向进行操作。即使你的数组不容易实现多态，你仍然可以像这样创建一个 `Applicative` 实例。

数据 `ArrayLike x = MkAL {sizeOf :: Int, eltOf :: Int -> x}`

实例 `Applicative ArrayLike` 其中 `pure x = MkAL maxBound (pure x) MkA`

`L i f <*> MkAL j g = MkAL (min i j) (f <*> g)`

爱好者会注意到，我已经将 `(Int ->)` 应用式与由 `(maxBound, min)` 单元生成的应用式进行了乘积运算。你能建立一个清晰的对应关系吗？

`imAL :: 图像 -> 类数组 浮点数` `allm :: 类数组 浮点数 -> 图像`

通过投影和制表吗？如果是的话，你可以编写代码 `like` 这个。

`allm $ (f <$> imAL a1 <*> ... <*> imAL an)`

而且，如果你接着想将这个模式封装成一个重载的操作符，

`imapp :: (浮点数 -> ... -> 浮点数) -> (图像 -> ... -> 图像)`

它是类型类编程中的一个标准练习！（如果需要更多提示，请提问。）然而，关键点在于，包装策略意味着你不需要改动数组结构就能在其上方构建函数式的上层结构。

## 4.16 这个简单的 Haskell 函数是否已经有一个众所周知的名称？（强度）

如果库中有 `(,) x` 的 `Traversable` 和 `Foldable` 实例（而且我想我也必须为它们的缺失承担一些责任）。...

`instance Traversable ((,) x) where traverse f (x, y) = (,) x <$> f y` `instance Foldable ((,) x) where foldMap = foldMapDefault ...` 那么这个（有时称为“strength”）将是 `Data.Traversable.sequence` 的一个特化。`sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)` 因此 `sequence :: (Monad m) => ((,) x) (m a) -> m (((,) x) a)` 即 `sequence :: (Monad m) => (x, m a) -> m (x, a)`

事实上，`sequence` 并没有真正使用 `Monad` 的全部能力：`Applicative` 就足够了。此外，在这种情况下，与 `x` 配对是线性的，因此 `traverse` 只会做 `<$>`，而不是 `pure` 与 `<*>` 的其他随机组合，并且（正如在其他地方已经指出的）你只需要 `m` 具有函子结构即可。

## 4.17 孩子们一切安好

并不是要否定将所有 `isRight` 作为问题的好答案，我会在某种程度上质疑这个问题。计算一个布尔值，判断列表中的所有 `Either` 值是否都是 `Right`，这样做有什么意义呢？它能让你做什么？一个答案是，它让你能够从整个列表中去掉 `Right` 标签，将整个列表视为没有错误。

一个更具信息量的选项可能是构造一个类型为

```
[Either String Int] -> Either String [Int]
```

以便你得到所有未标记的整数或与第一个烦人的左侧相关联的消息，而不是仅仅得到一个真或假的值。

有一个标准函数可以做到这一点（以及许多其他事情）。它利用了列表作为一种数据结构，具有标准的遍历模式，并且 `Either String` 编码了错误管理计算的概念，具有标准的失败和成功传播模式。这个类型已经完成了艰难的工作。你所需要做的就是。。。

序列A

## 4.18 为什么 `((,) r)` 是一个函子，却不是一个应用函子？

假设我们有

```
纯粹 :: 对所有 r a. a -> (r, a)
```

然后，特别是，我们有

```
魔法 :: 对于所有 r. r 魔法 = fs
t (pure ())
```

现在，我们可以将类型变量 `r` 专化为得到

```
魔法 :: 虚无
```

其中 `Void` 是没有构造函数的数据类型，这意味着

```
魔法 = 未定义
```

但是由于类型变量（以及专门化它们的类型）在运行时不发挥作用，这意味着魔法是 *always* 未定义的。

我们发现，`((,) r)` 只有在 *inhabited* `r` 的情况下才能是 `Applicative`。还有更多内容。对于任何这样的实例，我们可以写作

```
munge :: r -> r -> r
munge r0 r1 = fst ( pure (\ _ _ -> ()) <*> (r0, ()) <*> (r1, ()) )
```

定义一个二元运算符在 `r` 上。应用法则实际上告诉我们，`munge` 必须是一个结合运算符，能够吸收两边的 `magic`。

也就是说，*is* 存在一个合理的实例

实例单群 `r => 应用函子 ((,) r)` 其中

```
纯 a = (mempty, a) (r0, f) <*> (r1, s) = (mappend r0 r1, f s)
```

（当你获取纯粹的=返回时，你得到的正是这个；`<*>` = `ap` 来自 `Monad (Writer r)`）。

当然，一些拘泥于细节的人可能会争辩说，定义是合法的（尽管无益的）。

实例 `Monoid r` 其中 `mempty = 未定义` `ma`  
`ppend __ = 未定义` -- `Monoid` 法则显然  
 成立

但我认为，任何合理的类型类实例都应当对语言中已定义的片段作出非平凡的贡献。

## 4.19 应用式重写（供读者）

作为一个无所事事的废物，我想我可以让计算机为我做扩展。因此，我在 `GHCi` 中输入了

```
let pu x = "(\\_ -> " ++ x ++ ")"
let f >*< a = "(\\g -> " ++ f ++ " g (" ++ a ++ " g))"
```

所以现在我有纯粹的和 `<*>` 的有趣版本，它们将看起来像表达式的字符串映射为看起来像更复杂表达式的字符串。然后我以类似的方式定义了 `sequenceA` 的类似物，用字符串替换函数。

```
让 sqa [] = pu "[]"; sqa (f : fs) = (pu "(:)" >*< f) >*< sqa fs
```

我随后能够生成该示例的展开形式，如下所示

```
putStrLn $ sqa ["(+3)", "(+2)"] ++ " 3"
```

其已正式印制

~~I'm sorry, it seems like your source text is incomplete. Could you please provide the full text you'd like translated?~~ ) g ( (\\\_ ->

此最后，复制到提示中，得出了

```
[6,5]
```

将我的“元程序”的输出与题目中的尝试进行比较，可以看到一个更短的 `lambda` 初始前缀，这是由于 `<*>` 操作的嵌套更浅所致。记住，它是

```
(pure (:) <*> (+3)) <*> ((pure (:) <*> (+2)) <*> pure [])
```

因此外部 `(:)` 应仅深度为三个 `lambda`。我怀疑提议的扩展可能对应于上述公式的另一种括号版本，也许是

```
pure (:) <*> (+3) <*> pure (:) <*> (+2) <*>          纯 []
```

确实，当我评估

```
输出 $ pu "(:)" >*< "(+3)" >*< pu "(:)" >*< "(+2)" >*< pu "[]" ++ " 3 "
```

我明白了

```
(\\g -> (\\g -> (\\g -> (\\g -> (\\_ -> (:)) g ((+3) g)) g ((\\_ -> (:)) g)) g ((+2) g)) g (
```

看起来它与（更新的）匹配

~~It looks like the source text might be incomplete or not properly formatted. Could you please provide the full source text for the (a~~

我希望这次机器辅助的调查有助于澄清发生了什么。

## 4.20 序列化对角化

不是每种类型都是可序列化的。如何在 `String -> String` 和 `String` 之间建立同构？如果你给我 `String -> String` 的 `Read` 和 `Show` 实例，我可以找到一个像这样不可序列化的函数：

```
邪恶 :: 字符串 -> 字符串
邪恶 s = 映射 succ (读取 s s
++ " 邪恶")
```

假设

读（显示邪恶）= 邪恶

我们得到

```
邪恶 (展示邪恶)
= 映射 succ (读取 (显示 恶意) (显示 恶意) ++ " 恶意") = 映射 succ (恶意
(显示 恶意) ++ " 恶意") = 映射 succ (恶意 (显示 恶意)) ++ "!fwjm"
```

因此，如果恶意（显示恶意）被定义，那么它有一个满足 `c = succ c` 的第一个字符 `c`，这是不可能的。

一般来说，函数是无法序列化的。有时，我们编写数据类型来封装函数，因此并非所有数据类型都是可序列化的。例如，

```
数据精神科医生
= 听（字符串 -> 精神科医生）| 收费 费用
```

有时候，即使是对于这些类型，你也可能选择提供部分实现的 `Read`（某些情况缺失）和 `Show`（例如，使用占位符或函数的制表符），但没有公认的方式来选择它们，也没有理由预期两者都会存在。

正如其他人所提到的，严谨的序列化是 `Serialize` 的专属。我倾向于出于诊断目的使用 `Show` 和 `Read`，尤其是在 `ghci` 中尝试一些东西。就此目的而言，`Show` 无疑更为有用，因为 `ghci` 有一个 `Haskell` 解析器来进行读取。

## 4.21 中缀运算符的应用式风格？

她让你写

```
(| a ++ (| b ++ c |) |)
```

如果有用的话。当然，引入预处理层会有一些开销。

## 4.22 Applicative 中的幺半群在哪里？

也许你正在寻找的单元是这个。

新类型 `AppM f m = AppM (f m)` 派生 `Show`

```
实例 (Applicative f, Monoid m) => Monoid (AppM f m) 其中 mempty = AppM (pure mempty)
mappend (AppM fx) (AppM fy) = AppM (pure mappend <*> fx <*> fy)
```

作为下面的评论所指出的，它可以在reducer库中找到，名为Ap。它是Applicative的基础，因此让我们来解析一下。

特别要注意的是，由于  $()$  显然是一个 Monoid， $\text{AppM } f ()$  也同样是一个 Monoid。而这正是潜藏在  $\text{Applicative } f$  背后的那个 Monoid。

我们本可以坚持将  $\text{Monoid } (f ())$  作为  $\text{Applicative}$  的超类，但那样会彻底搞砸事情。

```
> mappend (AppM [(0,0)]) (AppM [(0,0,0)])
AppM [(0,0,0,0,0,0)]
```

底层的  $\text{Applicative } []$  单元群是自然数的 *multiplication*，而列表的“显而易见”单元结构是连接，它专门化为自然数的 *addition*。

数学警告。依赖类型警告。伪 Haskell 警告。

理解正在发生什么的一种方式，是考虑那些在 Abbott、Altenkirch 和 Ghani 所说的依赖类型意义下恰好是 *containers* 的  $\text{Applicative}$ 。我们很快会在 Haskell 中拥有这些。我就假装未来已经到来。

```
数据 (<|) (s :: *) (p :: s -> *) (x :: *) 其中 (<|:) :: pi (a :: s) -> (p a -> x)
-> (s <| p) x
```

数据结构  $(s <| p)$  的特点是

- 形状  $s$ ，用于告诉你容器看起来是什么样的。
- 位置  $p$  告诉你 *for a given shape* 可以将数据放在哪里。

上述类型表示，为这样的结构提供数据就是选择一种形状，然后用数据填充所有位置。

$[]$  的容器表示是  $\text{Nat} <| \text{Fin}$ ，其中

```
数据 Nat = Z | S Nat 数据 Fin (n :: N
at) 其中 FZ :: Fin (S n) FS :: Fin n ->
Fin (S n)
```

因此  $\text{Fin } n$  恰好有  $n$  个值。也就是说，列表的形状是它的 *length*，这告诉你需要多少个元素来填满列表。

你可以通过取  $f ()$  来找到 Haskell Functor  $f$  的形状。通过使数据变得平凡，位置就不再重要。在 Haskell 中以通用方式构造位置的 GADT 要困难得多。

参数化性告诉我们，容器之间的多态函数在

```
forall x. (s <| p) x -> (s' <| p') x
```

请提供需要翻译的 Markdown 源文本。

- 一个函数  $f :: s \rightarrow s'$  将输入形状映射到输出形状
- 一个函数  $g :: \text{pi } (a :: s) \rightarrow p' (f a) \rightarrow p a$  映射（对于给定的输入形状）输出位置回到输入位置，其中输出元素将来自。

```
<!-->
```

```
形态 f g (a <|: d) = f a <|: (d . g a)
```

（私下里，那些接受过基本汉考克训练的我们也把“形状”看作“命令”，把“位置”看作“有效响应”。于是，容器之间的一个态射正好就是一个“设备驱动程序”。不过我跑题了。）

沿着类似的思路，要让一个容器成为  $\text{Applicative}$  需要什么？首先，

纯 ::  $x \rightarrow (s \leq | p) x$

等价地说

纯 ::  $(() \leq | \text{常量 } ()) x \rightarrow (s \leq | p) x$

那必须由  $\{v^*\}$  给出

$f :: () \rightarrow s \dashv s$  中的一个常量  $g :: \text{pi } (a :: ()) \rightarrow p(f()) \rightarrow \text{Const } () a \dashv$  平凡

其中  $f =$  对某些而言为常数中性

中性 ::  $s$

现在，怎么样

$(\leq^* \triangleright) :: (s \leq | p)(x \rightarrow y) \rightarrow (s \leq | p) x \rightarrow (s \leq | p) y$

? 再次，参数性告诉我们两件事。首先，用于计算输出形状的唯一有用数据是两个输入形状。我们必须有一个函数

输出形状 ::  $s \rightarrow s \rightarrow s$

其次，我们能够用一个  $y$  填充某个输出位置的唯一方式，是从第一个输入中选择一个位置以在 ‘ $x \rightarrow y$ ’ 中找到一个函数，然后从第二个输入中选择一个位置来获得它的参数。

$\text{inPos} :: \text{pi } (a :: s)(b :: s) \rightarrow p(\text{outShape } a \ b) \rightarrow (p \ a, p \ b)$

也就是说，我们总能确定决定某个输出位置输出的那一对输入位置。

应用法则告诉我们，`neutral` 和 `outShape` 必须遵守单元法则，而且，此外，我们可以按照以下方式提升单元：

$\text{mappend } (a :< | : f) (b :< | : g) = \text{outShape } a \ b :< | : \backslash z \rightarrow \text{let } (x, y) = \text{inPos } a \ b \ z \text{ in mappend } (f \ x) (g \ y)$

这里还有一些要说的内容，不过为此，我需要对容器上的两种操作进行对比。

组成

$(s \leq | p) \cdot (s' \leq | p') = ((s \leq | p) s') \leq | \backslash (a :< | : f) \rightarrow \text{Sigma } (p \ a) (p' \cdot f)$

其中 `Sigma` 是依赖对类型

数据 `Sigma`  $(p :: *) (q :: p \rightarrow *)$  其中 `Pair` ::  $\text{pi } (a :: p) \rightarrow q \ a \dashv \text{Sigma } p \ q$

那到底是什么意思？

- 你选择一个外形
- 你为每个外部位置选择一个内在形状
- 复合位置是外部位置和内部位置的组合，适用于位于该位置的内部形状。

或者，在汉考克

- 你选择一个外部命令
- 你可以等到看到外部响应后再选择内部命令
- 复合响应是先对外部命令作出响应，然后根据你的策略对内部命令作出响应。

或者，更直白地说

- 当你创建一个列表的列表时，内部列表可以具有不同的长度

结合一个 `Monad` 会扁平化组合。在它背后不仅仅是形状上的一个单元，而是一个 *integration* 运算符。也就是说，

连接  $:: ((s \leq p) \cdot (s \leq p)) x \rightarrow (s \leq p) x$

需要

积分  $:: (s \leq p) s \rightarrow s$

你的自由单子为你提供了策略树，在其中你可以利用一个命令的结果来选择策略的其余部分。就好像你是在与 1970 年代的电传打字机交互一样。

与此同时……

张量

两个容器的张量（也由于 Hancock）由以下公式给出：

~~Please provide the text you would like to have translated.~~  $(p \ a, p' \ b)$

也就是说

- 你选择两个形状
- 一个位置是一个位置对，每个形状对应一个位置。

或

- 你选择两个命令，且不查看任何响应
- 一个响应就是一对响应。

或

- $[] \times []$  是 *rectangular matrices* 的类型：‘内部’列表必须具有相同的长度

后者是为什么在 Haskell 中很难获取  $\times$  的线索，但在依赖类型的设置中却很容易的原因。

与复合类似，张量是一个以恒等函子为单位元的么半群。若将单子所基于的复合替换为张量，我们会得到什么？

纯  $:: \text{Id } x \rightarrow (s \leq p) x$  谜题  $:: ((s \leq p) \times (s \leq p)) x \rightarrow (s \leq p) x$

但这究竟会是什么神秘之物呢？这并不神秘，因为我们知道在容器之间构造多态函数有一种相当严格的方法。必然会有…

$f :: (s, s) \rightarrow s$

$g :: \text{pi } ((a, b) :: (s, s)) \rightarrow p (f (a, b)) \rightarrow (p \ a, p \ b)$

而那些正是我们之前所说决定了  $\times$  的因素。

`Applicative` 是由张量生成的带效应编程的概念，而 `Monad` 是由组合生成的。你不需要等待外部响应来选择内部命令，这就是为什么 `Applicative` 程序更容易并行化的原因。

看到  $[] \times []$  作为矩阵形式告诉我们，为什么  $\times$  对于列表来说是建立在乘法之上的。

自由应用函子是带有调节钮的自由单子。对于容器，

自由  $(s <| p) = [s] <| \text{所有 } p$

在哪里

所有  $p [] = ()$  所有  $p (x : xs) = (p x, \text{所有 } p xs)$

所以“命令”是一个大的命令列表，就像一副打孔卡片。你在选择卡片组之前看不到任何输出。“响应”是你的线路打印机输出。那是1960年代。

所以，事情就是这样。Applicative 的本质，张量而非组合，要求一个基础的单群，并且元素的重新组合必须与单群兼容。

## 4.23 由幺半群构造的 Applicative（包括 min 和 max）

Applicative [] 具有生成所有可能组合的行为，而不是任何类似拉链式（zippy）行为，其根本原因在于 Applicative 是 Monad 的一个超类，并且在存在 Monad 实例时，旨在按照该 Monad 实例的方式来表现。Monad [] 将列表视为失败与优先选择的机制，因此 Applicative [] 实例也是如此。人们经常使用 applicative 接口来重构 monadic 代码，以减少对中间值所需的命名数量，并增加并行化的机会。如果这会导致函数语义发生显著变化，那将会相当可怕。

撇开这一点不谈，事实是，对于 Applicative [] 实例，你的选择多得让人眼花缭乱；如果再考虑空/非空以及有限/余归纳/无限等变体，更是如此。为什么会这样？

嗯，正如我在这个回答中提到的，每个 Applicative f 在其诞生之初都是一个 Monoid (f ())，在我们开始担心 values 之前，先将数据的 shapes 组合起来。列表就是一个典型的例子。

[()] 基本上是数字的类型。数字在许多方面都是幺半群。

从 Monad [] 推导出 Applicative [] 等同于选择由 1 和 \* 生成的幺半群。

同时，Applicative ZipList 利用 Haskell 的共归纳合并，等价于选择由无穷大和最小值生成的单一元。

问题提出了一个不合法的实例，但接近于一个合法的实例。你会注意到 <\*> 在空函数列表中没有定义，但对于非空函数列表，它会填充以匹配参数列表。不对称地，当参数用完时，它会截断。似乎哪里不太对劲。

下面是两个候选修复方案。

一种方法是在两侧遇到 empty 时进行截断，然后你必须取 pure = 的 repeat，这样你就得到了 ZipList。

另一个方法是排除空列表并在两侧填充。然后，你会得到由 1 和 maximum 生成的 positive 数字上的 Monoid 生成的 Applicative。所以这根本不是 ZipList。这就是我在这个答案中称为 PadMe 的东西。你需要排除 0 的原因是，对于 <\*> 输出中的每个位置，你需要指向来自两个输入的位置，分别是函数及其参数所在的位置。如果没有可填充的内容，就无法填充。

这是一个有趣的游戏。选择一个数字上的单半群（Monoid），看看你能否将它发展成一个适用于列表的应用函子（Applicative）！



## 第五章

# 单子

### 5.1 为什么我们使用单一函数 $a \rightarrow m b$

从某种意义上说，你是对的。既然每个单子  $m$  都是一个函子，我们可以使用 `fmap f`，其中函数  $f :: a \rightarrow b$ ，将一个  $m a$  变成一个  $m b$ ，但这里有个问题。 $b$  是什么？

我喜欢把这样的  $m$  理解为表示“计划获取”，其中“计划”涉及某种超出纯计算的额外交互。如果你有一个“计划获取 `Int`”，并且你想要一个“计划获取 `String`”，你可以使用 `fmap` 以及一个位于 `Int -> String` 中的函数，但该函数的类型告诉你，从 `Int` 得到 `String` 并不涉及任何进一步的交互。

这并不总是这样：也许 `Int` 是学生注册号，而 `String` 是他们的名字，因此从一个转换到另一个的计划需要在某个表中进行外部查找。这样，我就没有从 `Int` 到 `String` 的纯函数，而是有一个从 `Int` 到“计划-获取 `String`”的纯函数。如果我将它映射到我的“计划-获取 `Int`”上，那也没问题，但我最终会得到“计划-获取（计划-获取 `String`）”，我需要将外部和内部计划合并。

总体情况是，我们拥有足够的信息来计算获取更多信息的计划。这正是  $a \rightarrow m b$  所建模的含义。具体来说，我们有 `return :: a -> m a`，它把我们已有的信息转换成一个不采取任何进一步行动、却恰好给出该信息的计划；并且我们有  $(\gg=) :: (a \rightarrow m b) \rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m c)$ ，它将两个这样的东西进行组合。我们还知道  $(\gg=)$  是结合的，并且在左右两侧都会吸收 `return`，就像在经典命令式编程中，`;` 是结合的并且会吸收 `skip` 一样。

使用这种组合式方法从较小的计划构建更大的计划更为方便，并且可以将“plan-to-get”层的数量保持为一致的 *one*。否则，你需要用 `fmap` 构建一个  $n$  层的计划，然后在外层执行恰当数量的 `join` 操作（这将成为该计划的脆弱属性）。

现在，由于 Haskell 是一种具有“自由变量”和“作用域”概念的语言，其中的  $a$  在

`Please provide the text you would like to have translated.`

表示“整体输入信息”可以简单地认为来自我们已经拥有的事物范围，而留下

```
(\gg=) ::      m b  -> (b -> m c) ->      m c
```

于是我们回到“`bind`”，它是一种以最符合程序员习惯的形式呈现组合结构的工具，类似于局部定义。

总而言之，你可以使用一个  $a \rightarrow b$ ，但通常你需要  $b$  表示“获取某物的计划”，如果你想以组合的方式构建计划，这是一个有用的选择。

待修复: <http://i.stack.imgur.com/7ts7a.jpg>

图 5.1: 树的树

抱歉, 我无法处理图片链接。如果你有文本内容需要翻译, 请提供文本, 我会为你翻译。

图 5.2: 在此输入图像描述

## 5.2 使用 Join 替代 Bind 的 Monad

不必深挖隐喻, 我想建议将一个典型的单子  $m$  理解为“产生  $a$  的策略”, 因此类型  $m\ value$  是一种一等的“产生一个值的策略”。不同的计算或外部交互观念需要不同类型的策略, 但这一通用观念需要某种规整的结构才能说得通:

- 如果你已经有一个值, 那么你就有一种产生一个值的策略 ( $return :: v \rightarrow m\ v$ ), 它除了产生你已经拥有的那个值之外不包含任何其他内容;
- 如果你有一个将一种值转换为另一种的函数, 你可以把它提升到策略 ( $fmap :: (v \rightarrow u) \rightarrow m\ v \rightarrow m\ u$ ), 只需等待策略交付其值, 然后对其进行转换;
- 如果你有一个用于产生一个值的策略的策略, 那么你就可以构造一个用于产生一个值的策略 ( $join :: m\ (m\ v) \rightarrow m\ v$ ), 它会遵循外层策略, 直到它产生内层策略, 然后一路遵循该内层策略直到得到一个值。

让我们举个例子: 叶标记二叉树。。。

数据  $Tree\ v = \text{叶 } v \mid \text{节点 } (Tree\ v) (Tree\ v)$

... 表示通过抛硬币来生成东西的策略。如果策略是  $Leaf\ v$ , 那么就得到  $v$ ; 如果策略是  $Node\ h\ t$ , 则抛硬币, 如果是“正面”则按照策略  $h$  继续, 如果是“反面”则按照策略  $t$  继续。

实例 Monad 树 其中  
返回 = 叶

一种产生策略的策略是一个带有树标签的叶子的树: 在每个这样的叶子的位置, 我们可以直接嫁接进它所标记的树。。。

$join\ (Leaf\ tree) = tree$   $join\ (Node\ h\ t) = Node\ (join\ h)\ (join\ t)$

... 当然, 我们还有  $fmap$ , 它只是重新标记叶子

Functor Tree 的实例, 其中

$fmap\ f\ (Leaf\ x) = Leaf\ (f\ x)$   $fmap\ f\ (Node\ h\ t) = Node\ (fmap\ f\ h)\ (fmap\ f\ t)$

这是一个产生策略以制定策略来产生  $I$  的策略。  $nt.$

掷一枚硬币: 如果是“正面”, 再掷一枚硬币以决定两种策略 (分别为“掷一枚硬币决定生产 0 或生产 1”或“生产 2”); 如果是“反面”, 则生产第三种 (“掷一枚硬币决定生产 3 或掷一枚硬币决定生产 4 或 5”)。

这显然衔接起来, 形成了一种产生  $I$  的策略  $nt.$

我们利用的事实是, “产生价值的策略”本身也可以被视为一种价值。在 Haskell 中, 策略作为值的嵌入是默默进行的, 但在英语中, 我使用引号来区分使用策略与仅仅讨论它。join 操作符表达了

抱歉，我目前无法访问图片链接。如果你能提供图片中的文本内容，我可以帮你翻译。

图 5.3: 用于计算的拼图块

策略“以某种方式制定并随后遵循一种策略”，或者“如果你正在 *told* 一种策略，那么你随后可以 *use* 它”。

(题外话。我不确定这种“策略”式的方法是否是一种足够通用的方式来思考单子以及值/计算的区分，还是它只是另一个糟糕的隐喻。我确实发现以叶子标注的树状类型是一个有用的直觉来源，这或许并不令人惊讶，因为它们是 *free* 单子，结构刚好足以成为单子，但也仅此而已。)

PS “bind” 的类型

```
(>>=) :: m v -> (v -> m w) -> m w
```

说“如果你有一个生成 *v* 的策略，并且对于每个 *v*，有一个后续策略生成 *w*，那么你就有一个生成 *w* 的策略。”我们如何通过结合来表达这一点？

```
mv >>= v2mw = join (fmap v2mw mv)
```

我们可以通过 *v2mw* 重新标记我们的 *v* 生成策略：不再为每个 *v* 值产出，而是产出随之衔接、随时可加入的 *w* 生成策略！

## 5.3 在列名单子中使用 return 与不使用 return

为了理解为什么会得到这些特定的结果，去糖化的解释非常有帮助。让我在此基础上再补充一些关于培养对 Haskell 代码理解直觉的一般性建议。

Haskell 的类型系统不区分两个可分离的 “*monadic*” 用途：

- *[x] values* 的类型，是从 *x* 中选取元素的列表
- *[x]* 允许优先选择的 *x* 的元素的 *computations* 类型

这两个概念具有相同的表示形式这一事实，并不意味着它们扮演着相同的角色。在 *f1* 中，*[x, x+1]* 扮演的是计算的角色，因此它所生成的可能性会被合并到整个计算生成的选择之中：这正是列名单子 (list monad) 的 *>>=* 所做的事情。然而，在 *f2* 中，*[x, x+1]* 扮演的是值的角色，因此整个计算会在两个值之间生成一个有优先级的选择（而这两个值恰好是列表值）。

Haskell 并不使用类型来做出这种区分 [你现在可能已经猜到我认为它应该这么做，但那是另一个话题]。相反，它使用语法。因此，你需要训练大脑，在阅读代码时区分值和值计算的角色。do 语法是一种用于构造 *computations* 的特殊语法。do 里面的内容是由以下模板构建的：

这三块蓝色的部分进行做计算。我已将计算孔标记为蓝色，值孔标记为红色。这不是完整的语法，仅是一个帮助你在脑海中理解代码片段的指南。

确实，你可以在提供的蓝色位置中写入任何旧的表达式，只要它具有合适的单子类型，由此生成的计算将按需要使用 *>>=* 合并到整体计算中。在你的 *f1* 示例中，你的列表位于一个蓝色位置，并被当作优先选择来处理。

同样地，你也可以在红色位置书写表达式，它们很可能具有单子类型（比如这里的列表），但它们仍然会被当作值来对待。这正是 *f2* 中发生的情况：可以这么说，结果的外层括号是蓝色的，而内层括号是红色的。

在阅读代码时训练你的大脑去区分“值”和“计算”，这样你就能本能地知道文本中的哪些部分在承担哪种职责。一旦你重新“编程”了自己的大脑，*f1* 和 *f2* 之间的区别就会显得完全正常！

## 5.4 示例展示单子不可组合

对于一个小的反例，考虑终端

单子.

数据 `Thud x = Thud`

返回和 `>>=` 只是发出“砰”的一声，法律保持平凡。现在我们还为 `Bool` 引入 `writer` 单子（假设具有异或单体结构）。

数据 `翻转 x = 翻转 布尔 x`

我 `instance Monad Flip where return x = Flip False x >>= f = f x Flip True x`  
`>>= f = Flip (not b) y where Flip b y = f x`

嗯，呃，我们需要组成

新类型 `(::) f g x = C (f (g x))`

现在尝试定义。。。

实例 `Monad (Flip :: Thud) where -- 这实际上是常量 'Bool' 函子 return x = C (Flip ??? Thud) ...`

参数性告诉我们，`???` 不能以任何有用的方式依赖于 `x`，因此它必须是一个常数。因此，`join . return` 必然是一个常数函数，因此该定律

连接 . 返回 = `id`

无论我们选择何种 `join` 和 `return` 的定义，都必然失败。

## 5.5 暂停单子

这是我使用 *free* 单子的方式。呃，嗯，它们是什么？它们是具有节点上动作和叶子上值的树，`>>=` 像是树接合。

数据 `f ::* x = Ret x | Do (f (f ::* x))`

在数学中，写  $F^*X$  来表示这样的事物并不罕见，因此我使用了这个古怪的中缀类型名称。要创建一个实例，你只需要 `f` 是你映射的东西：任何 `Functor` 都可以。

实例函子 `f => 单子 ((::*) f)` 其中返回 `= Ret Ret x >>= k = k x`  
 执行 `ffx >>= k = 执行 (fmap (>>= k) ffx)`

这仅仅是“在所有叶子上应用 `k`，并将结果嫁接到树上”。这些树可以表示 *strategies* 用于交互式计算：整棵树覆盖了与环境的所有可能交互，而环境选择在树中跟随哪个路径。它们为什么是 *free*？它们只是树，没有有趣的方程理论在其上，说明哪些策略与其他策略等价。

现在让我们做一个 `Functor` 的工具包，它对应一堆我们可能想要执行的命令。这个东西

数据 ( $\text{:>>:}$ )  $s\ t\ x = s\text{ :? } (t \rightarrow x)$

实例函子 ( $s\text{ :>>: }t$ ) 其中  $\text{fmap } k\ (s\text{ :? }f) = s\text{ :? } (k\ .\ f)$

刻画了在输入类型为  $s$ 、输出类型为  $t$  的 *one* 命令之后在  $x$  中获得一个值的思想。为此，你需要在  $s$  中选择一个输入，并说明在给定命令在  $t$  中的输出时，如何继续得到  $x$  中的值。要在这种结构上映射一个函数，只需将其附加到续延上。到目前为止，这都是标准配置。针对我们的问题，现在可以定义两个函子：

修改类型  $s = (s \rightarrow s) \text{:>>:} ()$  类型 产量 =  $()$   
 $\text{:>>:} ()$

就像我刚刚写下了我们希望能够执行的命令的值类型！现在让我们确保在这些命令之间可以提供一个 *choice*。我们可以展示选择函数对象会产生一个函数对象。更标准的设备。

数据 ( $\text{:+:}$ )  $f\ g\ x = L\ (f\ x) \mid R\ (g\ x)$

实例 (Functor  $f$ , Functor  $g$ )  $\Rightarrow$  Functor ( $f\text{ :+: }g$ ) 其中  
 $\text{fmap } k\ (L\ fx) = L\ (\text{fmap } k\ fx)\ \text{fmap } k\ (R\ gx)$   
 $= R\ (\text{fmap } k\ gx)$

因此，修改  $s\text{ :+: Yield}$  代表修改与让步之间的选择。任何 *signature* 简单命令（通过值而不是操作计算与世界进行交流）都可以通过这种方式转化为函子。必须手动进行这操作真是麻烦！

这就给了我你的 Monad：基于 *modify* 和 *yield* 的签名的自由 Monad。

暂停  $s = (\text{:*})$  (修改  $s\text{ :+:}$  产出)

我可以将 *modify* 和 *yield* 命令定义为 *one-do-then-return*。除了协商 *yield* 的虚拟输入之外，这只是机械操作。

$\text{mutate} :: (s \rightarrow s) \rightarrow \text{Pause } s\ ()$   
 $\text{mutate } f = \text{Do } (L\ (f\text{ :? } \text{Ret}))$

产量 :: 暂停  $s\ ()$  产量 = 做 ( $R\ ()\text{ :?}$  返回))

阶跃函数随后为策略树赋予了意义。它是一个 *control operator*，通过一个计算（可能）构建另一个计算。

步骤 ::  $s \rightarrow \text{暂停 } s\ () \rightarrow (s, \text{也许} (\text{暂停 } s\ ()))$  步骤  $s\ (\text{Ret } ()) = (s, \text{无})$  步骤  $s\ (\text{Do } (L\ (f\text{ :? } k))) = \text{步骤 } (f\ s)\ (k\ ())$  步骤  $s\ (\text{Do } (R\ ()\text{ :? } k)) = (s, \text{仅 } (k\ ()))$

*step* 函数运行该策略，直到它要么以 *Ret* 结束，要么 *yield*，并在此过程中不断变更状态。

一般方法如下：将 *commands* (按值交互) 与 *control operators* (操作计算) 分开；在命令的签名上构建“策略树”的自由单子（旋转手柄）；通过在策略树上递归实现控制操作符。

## 5.6 Haskell 单子 return 任意数据类型

为这种类型定义 Monad 实例的一种方式是将其实视为一个 *free monad*。实际上，这将  $A\ a$  看作是一种带有一个二元运算符  $C$  的小型语法，而变量由通过  $B$  构造器嵌入的  $a$  类型的值来表示。这使得  $\text{return}$  成为  $B$  构造器，用于嵌入变量，而  $>>=$  则是执行替换的运算符。

实例 Monad A 其中 返回 =  $B\ B\ x\ >>= f = f\ x\ C\ l\ r$   
 $>>= f = C\ (l\ >>= f)\ (r\ >>= f)$

不难看出， $(>>= B)$  执行恒等替换，并且替换的复合是结合的。

另一种更“命令式”的方式来看待这个单子，是认为它捕捉了这样一种计算的思想：这些计算可以掷硬币（或读取比特流，或以其他方式访问一系列二元选择）。

数据 硬币 = 正面 | 反面

任何可以翻转硬币的计算必须停止翻转并成为一个值（带  $B$ ），或者翻转硬币并继续进行（带  $C$ ），如果硬币是正面则以一种方式进行，如果是反面则以另一种方式进行。翻转硬币并告诉你结果的单子运算是

硬币 :: 一枚硬币 =  $C\ (B\ \text{正面})\ (B\ \text{反面})$

$A$  的  $>>=$  现在可以被视为对抛硬币计算进行序列化，使得后续计算的选择可以依赖于先前计算所给出的值。

如果你拥有一条无限的硬币流，那么（除了你非凡的好运之外），你也足够幸运，能够将任何  $A$ -计算运行到其值，如下所示

数据流  $x = x\ :\>\ \text{流}\ x\ \text{--}$  实际上，我的意思是“协同数据”

抛掷 :: 硬币流  $\rightarrow A\ v\ \rightarrow v$  抛掷  $\_ (B\ v) = v$  抛掷 (正面  $\rightarrow cs) (C\ h\ t)$   
 $= \text{抛掷}\ cs\ h\ \text{抛掷}\ (\text{反面}\ :\>\ cs) (C\ h\ t) = \text{抛掷}\ cs\ t$

这类 monad 的一般模式是：有一个用于返回值的构造器（这里是  $B$ ），以及一堆其他构造器，用来表示可能操作的选择，以及在给定某个操作结果时计算可以继续进行的不同方式。这里  $C$  没有非递归参数并且有两个子树，所以我可以判断一定只有一个操作，而且它只有两种可能的结果，因此就是掷硬币。

所以，它是用带有变量和一个二元运算符的语法进行替代，或者它是一种排列计算的方式，这些计算会抛硬币。哪个观点更好呢？嗯……它们是同一个硬币的两面。

## 5.7 我应该避免使用 Monad fail 吗？

一些单子具有合理的失败机制，例如终结单子：

数据失败  $x = \text{失败}$

有些单子（monad）没有合理的失败机制（未定义的并不合理），例如初始单子：

数据 返回 x = 返回 x

从这个意义上讲，强制所有的单子都必须有一个 `fail` 方法显然是一个缺陷。如果你在编写抽象于单子的程序 (`Monad m`)  $\Rightarrow$ ，那么使用那个通用的 `m` 的 `fail` 方法并不是很健康。那会导致你定义一个可以用单子实例化的函数，而 `fail` 在这种单子中其实并不应该存在。

我看到在使用失败时（尤其是间接地，通过匹配 `Pat <- 计算`）提出的反对意见较少，尤其是在处理一个已经明确指定了良好失败行为的特定 `monad` 时。这类程序希望能够在回归到旧的规范时继续存在，在那里非平凡的模式匹配会要求 `MonadZero`，而不仅仅是 `Monad`。

有人可能会争辩说，更好的做法是始终明确处理失败情况。我反对这个观点，理由有两个：（1）单子编程的目的是避免这种冗余，和（2）目前对单子计算结果进行案例分析的符号表示非常糟糕。SHE 的下一个版本将支持这种符号表示（该符号也出现在其他变种中）。

```
案例 <- 计算 Pat_1 -> 计算_1 ... P
at_n -> 计算_n
```

这可能会有一点帮助。

但整个情况真是一个糟糕的混乱。通常，通过支持的操作来表征单子（`monads`）是很有帮助的。你可以将 `fail`、`throw` 等视为某些单子支持的操作，而其他单子则不支持。Haskell 使得支持操作集合中的小范围本地化变化变得相当笨拙和昂贵，它通过解释如何在旧操作的基础上处理新操作来引入新操作。如果我们真心想要做得更好，我们需要重新思考 `catch` 是如何工作的，使它成为 *different* 本地错误处理机制之间的转换器。我经常想要将一个可能会无信息失败的计算（例如，通过模式匹配失败）用一个处理器括起来，该处理器在传递错误之前增加更多的上下文信息。我不禁觉得，有时候做到这一点比应该的要困难。

所以，这是一个还有改进空间的问题，但至少应当只在提供合理实现的特定单子中使用 `fail`，并且要正确处理这些“异常”。

## 5.8 为什么 Kleisli 不是 Monoid 的一个实例？

在图书馆设计的业务中，我们面临一个选择点，我们选择在我们的集体政策（或缺乏政策）中不完全一致。

`Monad`（或 `Applicative`）类型构造子的 `Monoid` 实例可以通过多种方式产生。逐点提升始终是可用的，但我们并不定义

```
我实例 (应用函子 f, 单位元素 x) => 单位元素 (f x) {- 其实并非如此 -} 其中 mempty = pure mempty map
pend fa fb = mappend <$> fa <*> fb
```

请注意，实例 `Monoid (a -> b)` 只是一个逐点提升，因此，`(a -> m b)` 的逐点提升确实会发生，每当 `m b` 的 `monoid` 实例对 `b` 上的 `monoid` 进行逐点提升时。

我们通常不进行逐点提升，不仅因为这会阻止其他承载类型恰好是应用类型的 `Monoid` 实例，还因为 `f` 的结构通常被认为比 `x` 的结构更为重要。一个关键例子是 *free monoid*，更常见的名称是 `[x]`，它是通过 `[]` 和 `(++)` 成为 `Monoid`，而不是通过逐点提升。单射结构来源于列表包装，而非被包装的元素。

我偏好的经验法则确实是优先考虑类型构造器中固有的单模结构，而不是逐点提升或特定实例化类型的单模结构，比如 `a -> a` 的组合单元。这些可以并且确实会得到新类型的封装。

关于当二者都存在时 `Monoid (m x)` 是否应当与 `MonadPlus m` 一致 (`Alternative` 也是类似)，一直存在争论。我的看法是，唯一好的 `MonadPlus` 实例只是一个 `Monoid` 实例的拷贝，但也有人并不这么认为。尽管如此，这个库在这一点上并不一致，尤其是在……（许多读者可能已经预见到我这个老问题）。……

……`Maybe` 的 `Monoid` 实例，它忽略了我们通常使用 `Maybe` 来建模可能失败这一事实，而是注意到：向数据类型中塞入一个额外元素的同一思想，可以用来在一个原本没有单位元的半群中赋予它一个单位元。这两种构造产生了同构的类型，但在概念上并不相关。（编辑：更糟的是，这个想法的实现也很别扭，把实例约束成了 `Monoid`，而实际上只需要 `Semigroup`。我希望能看到“`Semigroup` 扩展为 `Monoid`”的想法被实现，但 *not* 用于 `Maybe`。）

具体回到 `Kleisli`，我们三个显而易见的候选实例：

1. 具有 `return` 和 `Kleisli` 组合的 `Monoid (Kleisli m a a)`
2. `MonadPlus m => Monoid (Kleisli m a b)`，将 `mzero` 和 `mplus` 逐点提升到 `->`
3. `Monoid b => Monoid (Kleisli m a b)`，先将 `b` 的么半群结构经由 `m` 提升，然后 `->`

我认为还没有做出选择，只是因为不清楚该做出哪个选择。我犹豫着说，但我的投票会支持2，优先考虑来自 `Kleisli m a` 的结构，而不是来自 `b` 的结构。

## 5.9 提示中的单子？

按目前的情况来看，IO 特有的行为依赖于这样一种方式：IO 动作在某种程度上是类似状态的，并且是不可回退的。因此你可以说出类似这样的内容

```
s <- 读取文件 "foo.txt"
```

并获得一个实际值 `s :: String`。

很明显，维持这种互动不仅仅依赖于 `Monad` 结构。使用起来不会这么简单。

```
n <- [1, 2, 3]
```

说明 `value n` 拥有什么。

人们当然可以设想对 `ghci` 进行改造，使其打开一个提示符，允许通过多次命令行交互以 `do` 风格构建一个单子计算，并在关闭提示符时交付整个计算。但并不清楚检查中间值究竟意味着什么（当然，除非是为当前活动的单子 `m` 生成类型为 `m (IO ())` 的打印计算集合）。

但有意思的是，可以问一问：IO 有什么特殊之处，使得良好的交互式提示行为成为可能，这些特性是否可以被抽离并加以泛化。我不禁嗅到一丝关于“在提示下进行交互”的 `comonadic` 的“带上下文的值”叙事气息，但我还没有真正把它追踪清楚。或许可以通过思考这样一件事来处理我那个列表的例子：拥有一个指向可能值空间的 `cursor` 意味着什么，就像 IO 由于现实世界的此时此地而被强加了一个光标一样。感谢你提供的这些发人深省的思考素材。

## 5.10 这是一个应该使用 `liftM` 的情况吗？

`liftM` 和它的作用是将 `pure` 函数提升到单子（`monadic`）环境中，类似于 `Applicative` 组合子所做的。

```
liftM :: Monad m => (s -> t) -> m s -> m t
```

你尝试的代码有两个问题。其中一个只是缺少括号。

```
liftM sendAllTo :: IO 套接字 -> IO (字节串 -> 套接字地址 -> IO ())
```

这并不是你的本意。另一个问题是

```
sendAllTo :: Socket -> ByteString -> SocketAddr -> IO ()
```

这是一个 *monadic* 操作，因此对其进行提升将产生两层 IO。通常的方法是对应用的纯前缀加上括号，如下所示

```
liftM (sendAllTo s datastring) :: IO SocketAddr -> IO (IO ())
```

然后你可以使用 liftM2 来构建参数。

```
liftM2 SocketAddrInet ioport (inet_addr host) :: IO SocketAddr
```

这给你

```
liftM (sendAllTo s datastring) (liftM2 SocketAddrInet ioport (inet_addr host)) :: IO (IO ())
```

照现在这样，它将什么也实现不了，因为它解释了如何计算一个操作，却并没有真正调用它！这就是你需要

```
加入 (liftM (sendAllTo s datastring) (liftM2 SocketAddrInet ioport (inet_addr host))) :: IO ()
```

或者，更简洁地说

```
sendAllTo s datastring = << liftM2 SocketAddrInet ioport (inet_addr host)
```

插头。Strathclyde Haskell 增强版支持习惯用法括号，其中

```
([f a1 .. an] :: m t 如果 f :: s1 -> ... -> sn -> t 且 a1 :: m s1 ... an :: m sn。
```

这些对 Applicative m 起到的作用与 liftM 系列对 monad 所起的作用相同：将 f 视为一个纯的 n 元函数，而将 a1..an 视为具有效果的参数。Monad 也可以并且应该是 Applicative，因此

```
(| SocketAddrInet ioproto (inet_addr host) |) :: IO SocketAddr
```

和

```
(| (sendAllTo s datastring) (| SocketAddrInet ioproto (inet_addr host) |) |) :: IO (IO ())
```

该记法随后允许你像上面那样，通过后缀的 @ 来调用已计算的单子计算。

```
(| (sendAllTo s 数据 string) (| SocketAddrInet ioproto (inet_addr host) |) @ |) :: IO ()
```

注意，我仍然对应用的纯前缀加上括号，因此模板中的 f 就是整个 (sendAllTo s datastring)。这种记法允许你在任意位置用 标记纯参数，因此你可以这样写

```
Please do this: liftM2 (sendAllTo s datastring) (SocketAddrInet ioproto (inet_addr host)) @ |) :: IO () 如果你乐意的话。
```

牢骚。我们在弄清楚正确的 liftM、join、=<<、do，以及 (|...@|) 这些标点上花费了太多精力，只是为了说明如何在一个效果解释的上下文（这里是 IO）中，把一个类型切分为一个用于解释值的内核（这里是 ()）。如果这种向上切分能在类型中更明确地表达出来，我们的程序里就不需要那么多杂音来把值和计算硬性对齐了。我更希望计算机去推断 和 @ 标记该放在哪里，但目前而言，Haskell 的类型作为一种文档过于含糊，难以做到这一点。

### 5.11 Zappy 列表确实 *not* 形成了一个单子

我只是想澄清一下，带有“习语”标题并包含练习的版本其实是这篇论文的一个相当 *earlier* 草稿，最终出现在 JFP 上。那时，我错误地认为 *colists*（指可能是无限的，也可能是有限的列表）是一种 *Monad*，与 *zapp* 对应：有一个合理的 *join* 候选（在其他答案中提到过），但 Jeremy Gibbons 友好地指出，这并不满足 *Monad* 法则。反例涉及“参差不齐”的列表，列表内的长度各不相同。相应地，在 JFP 文章中，我们纠正了这一点。（我们对这一点感到相当高兴，因为我们喜欢发现那些其 ( $<*>$ ) 不是 *Monad* 的 *ap* 的应用函子。）

必然无限的列表（即 *streams*），通过排除不规则情况，确实形成了一个单子，其 *ap* 的行为类似于 *zapp*。作为提示，请注意，*Stream x* 与  $\text{Nat} \rightarrow x$  同构。

对造成的混淆深表歉意。有时把旧的、未完成的草稿（错误百出）随意放（哈哈）在网上是很危险的。

### 5.12 Haskell *io-streams* 和 *forever* 不会向 *std-out* 输出任何内容

你代码中 *forever* 的意思是，它需要永远才能在流中选择 *each* 值。它的类型

永远 :: *Monad m* => *m a* -> *m b*

是一个重要线索，表明使用 *forever* 构建的计算永远不会返回一个值：*forever* 的调用者可以任意选择类型 *b*，因此没有程序可以真正承诺返回该类型的值。这也是为什么你的程序通过类型检查的原因。你传递给 *forever* 的计算会反复执行以产生副作用（在这种情况下，选择一个随机数），但永远不会返回任何值，因此流永远不会开始运行。

你不需要用 *forever* 来生成一个持续不断的流。*makeInputStream* 的行为是在每次从流中请求一个值时运行其参数计算，因此你已经在这里获得了所需的重复。

## 第6章

# 微积分与类型

### 6.1 查找列表中某个元素的前一个元素

我最喜欢的一个被低估的工具对这种问题非常有用。让我拥有反向列表，这样我就不需要反转我的思维。

数据 `Bwd x = B0 | Bwd x :< x -- 最右边的是最近的`

把列表元素想象成算盘上一根算珠杆上的珠子。把其中几个拨到左边，并让你的手指停在紧挨着的下一个上。你得到了什么？手指左侧的一串珠子（最靠近的是最右边那个）、手指右侧的一串珠子（最靠近的是最左边那个），以及你手指按着的那颗珠子。

也就是说，列表的 *one-hole element context* 由洞两侧的前向和后向列表的对组成。

类型 `ListContext x = (Bwd x, [x])`

那些知道我旧歌的人会认出 `ListContext` 是 `[]` 的 *derivative*。  
焦点中的元素（你手指在珠子上）是

类型 `ListFocus x = (ListContext x, x)`

有一个有用的操作，它为每个列表元素添加上下文，使其聚焦。

聚焦 `:: [x] -> [ListFocus x]` 聚焦 = `go B0` 其中 `go xz [] = []` `go xz (x : xs)`  
= `((xz, xs), x) : go (xz :< x) xs`

例如，

焦点 `[1,2,3] = [((B0,[2,3]),1), ((B0 :< 1,[3]),2), ((B0 :< 1 :< 2,[]),3)]`

现在，要回答所有涉及某个元素及其直接周边的各种问题都变得非常容易了。你也许不会仅仅为了解决这个问题而去构造一个 `focus`，但这是我一直保留着的那种东西，因为它能解决很多问题。

`[p | ((_ :< p, _), q) <- focus xs, q == x]`

计算所有位于 `xs` 中 `x` 左侧的 `p` 值。As you can see.

(顺便一提，这个 `focus` 操作并非凭空而来。它源自列表这一数据类型的微分结构。这个回答（其中它被称为 `picks`）更详细地讲述了列表的故事，而这个回答则展开了数据类型泛型的故事。）

## 6.2 列表拆分

*This answer is in two parts. The first part addresses the question directly. The second part goes off on a tangent (literally) in order to dig about in the mathematics behind the first part: it may thus prove to be difficult material of limited interest, but I thought a few extremists might enjoy it.*

我迄今为止看到的答案巧妙地使用了列表推导式或其单子等价物，但它们使用 *equality* 来排除重复项，从而需要额外的 *Eq* 约束。这是一个解决方案，它生成两个不同 *positions* 中所有元素的配对。

首先，我写了一个方便的函数，它将列表的每个元素装饰为其他位置元素的列表：所有“选一个并留下其他”的方式。每当列表用于收集不重复选择的内容时，这非常有用，而这是我发现自己经常使用的。

```
选择 :: [x] -> [(x, [x])] 选择 [] = [] 选择 (x : xs) = (x, xs) : [(y, x : ys) | (y, ys) <- 选择 xs]
```

注意，`map fst . picks = id`，因此结果中每个位置所选取的元素都是原始列表中该位置的元素：这正是我所说的“装饰”。

现在使用与其他答案中相同的列表推导方法，选择两个变得容易。但我们可以从其选择中选择第二个组件，而不是从列表本身选择第一个组件，同时获取第二个组件的候选列表。

```
allPairs :: [x] -> [(x, x)] allPairs xs = [(y, z) | (y, ys) <- picks xs, z <- ys]
```

获取三元组就像两次选择一样简单。

```
allTriples :: [x] -> [(x, x, x)]
allTriples ws = [(x, y, z) | (x, xs) <- 从 ws 中选取, (y, ys) <- 从 xs 中选取, z <- ys]
```

为了保持一致性，几乎会让人想把代码写得稍微低效一些，在两处都写成 `(z, ) <- 选取 ys`，而不是 `z <- ys`。

如果输入列表没有重复项，那么输出中不会出现任何重复的元组，因为这些元组从不同的位置取元素。然而，你将会得到

```
选择> allPairs ["猫", "猫"] [("猫", "猫"), ("猫", "猫")]
```

为避免这种情况，可以随意使用 `allPairs . nub`，它会在选择之前移除重复项，并再次要求元素类型具有 *Eq* 实例。 *For extremists only: containers, calculus, comonads and combinatorics ahoy!* -----

`picks` 是一个更一般构造的一个实例，它源自微分学。一个有趣的事实是，对于任意给定的容器型函子 *f*，它的数学导数  $\partial f$  表示去掉一个元素的 *f*-结构。例如，

```
newtype Trio x = Trio (x, x, x) -- x^3
```

有导数

```
data DTrio x = Left3 ((), x, x) | Mid3 (x, (), x) | Right3 (x, x, ()) -- 3*x^2
```

可以将多种操作与此构造关联。假设我们可以真正使用  $\partial$  (，并且我们可以通过使用类型族)来编写代码。然后我们可以说

```
数据 InContext f x = (:-) {selected :: x, context ::  $\partial f$  x}
```

给出一种由上下文修饰的所选元素的类型。我们当然应该期望有这一操作

插入 :: InContext f x -> f x -- 将元素放回原位

这个 plug 操作会在我们对一棵树进行 zipper 操作时把我们朝向根移动，在这种树中，节点被视为子树的容器。我们也应当期望 InContext f 是一个余单子（comonad），并具有

counit :: InContext f x -> x counit = 选定

将所选元素投影掉并

cojoin :: InContext f x -> InContext f (InContext f x)

为每个元素添加其上下文，展示你可以 refocus 的所有可能方式，选择不同的元素。

不可估量的彼得·汉考克曾经建议我，我们还应该预期能够向“下”移动（意味着“远离根源”），收集从整个结构中选择上下文元素的所有可能方式。

picks :: f x -> f (InContext f x)

应该用其上下文来标注输入 f-structure 中的每一个 x 元素。我们应该期望的是

fmap 选择 . 挑选 = id

这是我们之前提到的定律，但也

fmap 插件（选择 fx）= fmap（const fx）fx

告诉我们，每个装饰元素都是原始数据的分解。我们在上面没有这个法则。我们有

picks :: [x] -> [(x, [x])]

装饰每个元素时，并不完全与其上下文相符：仅从其他元素的列表中，你无法看到“空洞”在哪里。事实上，

$\partial[] \ x = ([x], [x])$

将空洞之前的元素列表与空洞之后的元素分离开来。可以说，我本该写成

选择 :: [x] -> [(x, ([x], [x]))] 选择 [] = [] 选择 (x : xs) = (x, ([], xs)) : [(y, (x : ys, ys')) | (y, (ys, ys')) <- 选择 xs]

这当然也是一个非常有用的操作。

但真正发生的事情其实相当合理，只是有一点点不严谨。在我最初写的代码中，我在局部把 [] 用来表示 *finite bags* 或 *unordered lists*。Bags 是一种没有特定位置概念列表，因此如果你选取其中一个元素，它的上下文就是其余元素所组成的 bag。确实如此

$\partial \text{Bag} = \text{Bag}$  —— 真的？为什么？

因此，关于 `picks` 的正确概念确实是

选择  $:: \text{Bag } x \rightarrow \text{Bag } (x, \text{Bag } x)$

将袋子表示为 `[]`，这就是我们所得到的。此外，对于袋子，插头就是 `(:)`，并且在袋子相等（即排列）的情况下，选择的第二定律 *does* 成立。

从另一个角度看，袋（bag）可以被视为一个幂级数。一个袋是对任意大小的元组的选择，其中所有可能的排列（大小为  $n$  时有  $n!$  种）都被视为相同。因此，我们可以在组合学意义上把它写成一个按阶乘取商的幂的巨大求和，因为你必须将  $x^n$  除以  $n!$ ，以说明你选择这些  $x$  的  $n!$  种顺序都会得到同一个袋。

$$\text{Bag } x = 1 + x + x^2/2! + x^3/3! + \dots$$

所以

$$\partial \text{Bag } x = 0 + 1 + x + x^2/2! + \dots$$

将级数横向平移。事实上，你很可能已经认识到 `Bag` 的幂级数与 `exp`（或  $e^x$ ）的幂级数相同，而它因其导数仍为自身而著名。

所以，呼！这就是结果。一个从指数函数的数据类型解释中自然产生的操作，它是其自身的导数，是解决基于不放回选择问题的得力工具。

### 6.3 nub 作为列表推导式

@amalloy 的评论指出，列表推导式被限定在一种“局部”的视角中，这是这里的关键洞见。确实有一种合理的方法可以用列表推导式来编写 `nub`，但你首先需要改变你的视角。

一个经常有用的函数，遗憾的是在库中被省略了，它是将列表中每个元素与其上下文进行装饰的函数。

挑选  $:: [x] \rightarrow [(x, x, [x])]$  挑选 `[]` = `[]` 挑选  $(x : xs) = ((), x, xs) : [(x : bs, y, as) \mid (bs, y, as) <- \text{挑选 } xs]$

所以

选择 `[1,2,3]` = `[([],1,[2,3]), ([1],2,[3]), ([1,2],3,[])]`

每个列表元素都被放置在一个三元组的中间，左边是‘之前’的元素，右边是‘之后’的元素。

这个我的答案解释了使得在某种意义上“标准”操作的深层结构，这个操作可以从列表的结构中推导出来。但我们在使用它时并不需要那些背景信息。

`picks` 函数正好提供了我们编写 `nub` 作为列表推导所需的上下文信息。我们需要做的就是挑选出那些不出现在它们自己“前置列表”中的元素。

`myNub :: Eq x => [x] -> [x]` `myNub xs = [x \ (bs, x, as) <- picks xs, not (elem x bs)]`

我不对这一操作的效率作任何保证，但我确实喜欢将列表推导式与额外的空间上下文结合所带来的清晰性。

## 6.4 如何将二叉树拉链化为一个 Comonad 实例?

在微分学中，莱布尼茨的符号比牛顿的符号引起的混淆更少，因为它明确指出我们对哪个变量进行微分。事物中的上下文是通过微分给出的，因此我们必须注意正在被赋予上下文的内容。在这里，有两个“子结构”的概念在起作用：*subtrees* 和 *elements*。它们各自具有不同（但相关）的“上下文”概念，因此也有不同的“拉链”概念，其中拉链是事物和其上下文的配对。

你的 BTZ 类型被表述为 *subtrees* 的拉链 (zipper) 这一概念。然而，拉链的余单子 (comonad) 构造作用于 *elements* 的拉链：extract 的含义是“给出当前位置的元素”；duplicate 的含义是“用其上下文装饰每个元素”。因此你需要元素的上下文。令人困惑的是，对于这些二叉树，元素拉链与子树拉链是同构的，但这是出于一个非常特殊的原因（即它们构成了一个余自由余单子）。

通常，元素和子树拉链是不同的，例如，对于列表。如果我们从为列表构建元素拉链共单子开始，那么当我们回到树时，我们就不太可能迷失方向。我还想尝试填补更多的整体图景，不仅是为了你，也为了其他人。

子列表上下文

[a] 的 *sublist*-上下文仅由 [a] 给出，也就是从子列表走向整个列表的过程中我们经过的元素列表。在 [1,2,3,4] 中，[3,4] 的子列表上下文是 [2,1]。递归数据的子节点上下文总是列表，表示从该节点到根节点路径上所看到的内容。每一步的类型由针对递归变量，对单个数据节点的公式取偏导数来给出。因此这里

[a] = t 其中	-- t 是表示 [a] 的递归变量
t = 1 + a*t	-- 列表 a 要么是 [], 要么是 (a : t) 对
$\partial/\partial t (1 + a*t) = a$	-- 这是从节点到根的一步。
子列表上下文是 [a]	-- 此类步骤的列表

所以一个子列表拉链是一个对

```
数据 LinLZ a = LinLZ { subListCtxt
xt :: [a], subList :: [a] }
```

我们可以编写一个函数，将子列表重新插入其上下文，沿着路径反向回溯。

```
plugLinLZ :: LinLZ a -> [a] plugLinLZ (LinLZ { subListCtxt = [], subList = ys }) = ys plug
LinLZ (LinLZ { subListCtxt = x : xs, subList = ys }) = plugLinLZ (LinLZ { subListCtxt = xs
, subList = x : ys })
```

但是我们不能让 LinLZ 构成一个 Comonad，因为（例如）从

```
LinLZ { subListCtxt = [], subList = [] }
```

我们无法从 `LinLZ a` 中提取 *element* (an a, 只能提取 `subList` 列表元素上下文)

一个列表 *element* 上下文是一个由两个列表组成的对：焦点元素之前的元素和之后的元素。在递归结构中，元素上下文总是一个对：首先给出元素存储的子节点上下文，然后给出元素在其节点中的上下文。我们通过对节点的公式进行对变量求导来获得元素在其节点中的上下文，其中该变量代表元素。

[a] = t 其中	t 是代表 [a] 的递归变量
t = 1 + a*t	-- 列表中的 a 要么是 [], 要么是 (a : t) 对
$\partial/\partial a (1 + a*t) = t = [a]$	-- 头元素的上下文是尾部列表

所以元素上下文由一对给定。

t 类型  $DL\ a = [a]$  -- 元素所在节点的子列表上下文,  $[a]$  -- 元素所在节点的尾部 )  
(

并且一个元素拉链通过将这样的上下文与“在孔中”的元素配对来给出。

数据  $ZL\ a = ZL\ \{ \text{这个} :: a, \text{之间} :: DL\ a \}$  派生  
(显示, 相等, 函子)

你可以通过先重构该元素所在的子列表, 得到一个子列表拉链, 然后将该子列表插入其子列表上下文中, 把这样的拉链再变回一个列表 (从某个元素“向外”)。

```
outZL :: ZL a -> [a]
outZL (ZL { 此 = x, 介于 = (zs, xs) }) = plugLinLZ (LinLZ { subListCtxt = zs, subList = x : xs })
```

将每个元素置于上下文中  
给定一个列表, 我们可以将每个元素与其上下文配对。我们得到可以“进入”其中一个元素的方式列表。我们从这里开始,

```
进入 :: [a] -> [ZL a]
进入 xs = 更多进入 (LinLZ { 子列表上下文 = [], 子列表 = xs })
```

但真正的工作是由辅助函数完成的, 该函数作用于上下文中的列表。

```
moreInto :: LinLZ a -> [ZL a]
moreInto (LinLZ { subListCtxt = _, subList = [] }) = []
moreInto (LinLZ { subListCtxt = zs, subList = x : xs }) = ZL { this = x, between = (zs, xs) } : moreInto (LinLZ { subListCtxt = x : zs, subList = xs })
```

注意到输出反映了当前 `subList` 的形状。另外, `x` 的位置上的拉链包含这个 `= x`。此外, 用于装饰 `xs` 的生成拉链具有 `subList = xs` 和正确的上下文, 记录我们已经越过了 `x`。测试,

```
进入 [1,2,3,4] = [ ZL {this = 1, 在 = ([],[2,3,4]) }, ZL {this = 2, 在 = ([1],[3,4]) }, ZL {this = 3, 在 = ([2,1],[4]) }, ZL {this = 4, 在 = ([3,2,1],[]) } ]
```

列表元素拉链的共范畴结构

我们已经看到如何从一个元素出去, 或者进入一个可用的元素。共单结构告诉我们如何在元素之间移动, 要么停留在当前位置, 要么移动到其他的某个元素。

实例 `Comonad ZL` 其中

提取物为我们提供了我们正在访问的元素。

提取 = 这个

要复制一个拉链，我们将当前元素  $x$  替换为整个当前拉链  $zl$ （其中的这个 =  $x$ ）。

重复  $zl@(ZL \{ \text{这个} = x, \text{介于} = (zs, ys) \text{之间} \}) = ZL \{ \text{这个} = zl$

... 我们逐步分析上下文，展示如何在每个元素上重新聚焦。我们现有的 `moreInto` 让我们向内移动，但我们也必须向外移动。...

, 在 = 向外  $(LinLZ \{ \text{subListCtxt} = zs, \text{subList} = x : ys \})$ , `moreInto`  $(LinLZ \{ \text{subListCtxt} =$   
 $(x : zs, \text{subList} = ys \}))$  }

... 这涉及沿着上下文回溯，将元素移动到子列表中，如下所示

在外部  $(LinLZ \{ \text{subListCtxt} = [], \text{subList} = \_ \}) = []$  外部  $(LinLZ \{ \text{subListCtxt} = z : zs, \text{subList} = ys \}) = ZL \{ \text{this} = z, \text{between} = (zs, ys) \} : \text{外部} (LinLZ \{ \text{subListCtxt} = zs, \text{subList} = z : ys \})$

因此我们得到

d复制  $ZL \{ \text{this} = 2, \text{介于} = ([1],[3,4]) \} = ZL \{ \text{this} = ZL \{ \text{this} = 2, \text{介于} = ([1],[3,4]) \}, \text{介于} = [ ZL \{ \text{this} = 1, \text{介于} = ([],[2,3,4]) \} ], [ ZL \{ \text{this} = 3, \text{介于} = ([2,1],[4]) \}, ZL \{ \text{this} = 4, \text{介于} = ([3,2,1],[ ]) \} ] \}$

其中这是“停留在 2”，我们介于“移动到 1”和“移动到 3 或移动到 4”之间。

因此，共单子结构向我们展示了如何在列表中移动不同的 *elements*。子列表结构在找到元素所在的节点中起着关键作用，但复制的拉链结构是一个 *element* 拉链。

那么树木怎么样？

题外话：标记树已经是共单元了

让我重构你的二叉树类型，以突显一些结构。从字面上讲，让我们把标记为叶子或分叉的元素作为一个公共因子提取出来。我们还将隔离出解释这种叶子或分叉子树结构的函子 (TF)。

数据  $TF\ t = \text{叶子} \mid \text{分支} (t, t)$  派生 (Show, Eq, Functor) 数据  $BT\ a = a : \& TF (BT\ a)$   
 派生 (Show, Eq, Functor)

也就是说，每个树节点都有一个标签，无论它是叶子节点还是分叉节点。

无论何时我们有这样的结构：每个节点都有一个标签和一堆子结构，我们就有一个共单子：*cofree comonad*。让我再稍微重构一下，将 TF 抽象出来……

数据  $\text{CoFree } f \text{ } a = a : \& f (\text{CoFree } f \text{ } a)$  导出 (Functor)

因此，我们有一个一般的  $f$ ，在之前我们有  $\text{TF}$ 。我们可以恢复我们的特定树。

数据  $\text{TF } t = \text{叶子} \mid \text{分叉 } (t, t)$  派生 (显示, 等于, 函子) 类型  $\text{BT} = \text{CoFree } \text{TF}$  派生实例 显示  $a \Rightarrow \text{显示 } (\text{BT } a)$  派生实例 等于  $a \Rightarrow \text{等于 } (\text{BT } a)$

现在，我们可以一次性给出共自由共单态构造。由于每个子树都有一个根元素，每个元素都可以用其根所对应的树来装饰。

实例函子  $f \Rightarrow \text{共单模 } (\text{CoFree } f)$  其中  $\text{extract } (a : \& \_) = a$  -- 提取根元素  $\text{duplicate } t@(a : \& ft) = t : \& \text{fmap duplicate } ft$  -- 用整个树替换根元素

让我们举个例子

```
aTree = 0 :& 分叉 (1 :&
分叉 (2 :& 叶, 3 :& 叶)
, 4 :& 叶)
```

```
(复制 aTree = 0 :& 分叉 (1 :& 分叉 (2 :& 叶子, 3 :& 叶子), 4 :& 叶子)) :& 分叉 ((1 :& 分叉 (2 :& 叶子, 3 :& 叶子)) :& 分叉 ((2 :& 叶子) :& 叶子, (3 :& 叶子) :& 叶子), (4 :& 叶子) :& 叶子)
```

看到了吗？每个元素都与它的子树配对了！

列表不会产生共伴随代数，因为并非每个节点都有元素：特别是， $[]$  没有元素。在共伴随代数中，始终在当前位置有一个元素，并且你可以进一步查看树结构，*but not further up*。

在元素拉链余单子中，总有一个你所在的元素，并且你可以同时看到上方和下方。

二叉树中的子树和元素上下文，代数形式

$$d/dt \text{ (TF } t) = d/dt \text{ (1 + } t * t) = 0 + (1 * t + t * 1)$$

因此我们可以定义

类型  $\text{DTF } t = \text{Either } (), t) (t, ())$

就是说，“子结构的团块”内部的一棵子树要么在左边，要么在右边。我们可以检查“代入”是否可行。

```
plugF :: t -> DTF t -> TF t
plugF t (Left ((), r)) = Fork (t, r)
plugF t (Right (l, ())) = Fork (l, t)
```

如果我们实例化 `t` 并与节点标签配对，我们将得到一个子树上下文的步骤

类型 `BTStep a = (a, DTF (BT a))`

其与问题中的 `Partial` 是同构的。

```
plugBTinBT :: BT a -> BTStep a -> BT a
plugBTinBT t (a, d) = a :& plugF t d
```

因此，一个 `BT a` 位于另一个之内的 *subtree*-上下文由 `[BTStep a]` 给出。但是 *element* 上下文呢？嗯，每个元素都标记着某个子树，所以我们应当记录该子树的上下文以及由该元素标记的树的其余部分。 ld

数据 `DBT a = DBT`

```
{ below :: TF (BT a) -- 元素节点的其余部分, above :: [BTStep a] -- 元素节点的子树上下文 } deriving (Show, Eq)
```

烦人的是，我得自己实现一个 `Functor` 实例。

```
instance Functor DBT where fmap f (DBT { above = a, below = b }) = DBT { below = fmap (fmap f) b, above = fmap (f *** (either (Left . (id *** fmap f)) (Right . (fmap f *** id)))) a }
```

现在我可以说明什么是一个 *element zipper*。

```
data BTZ a = BTZ { here :: a, ctxt :: DBT a } deriving (Show, Eq, Functor)
```

如果你在想“有什么新的吗？”，你是对的。我们有一个上方的子树上下文，以及一个由此处及以下给出的子树。这是因为唯一的元素就是那些为节点加标签的元素。将一个节点拆分为一个元素及其上下文，等同于将其拆分为它的标签以及它的一团子结构。也就是说，这种巧合在共自由单子中成立，但一般情况下并不成立。

然而，这种巧合只是一个干扰！正如我们在列表的例子中看到的，我们并不需要元素拉链与子节点拉链相同，才能使元素拉链成为一个余单子。

按照与上面的列表相同的模式，我们可以为每个元素附加其上下文。这项工作由一个辅助函数完成，该函数会累积我们当前正在访问的子树上下文。

```
向下 :: BT a -> BT (BTZ a)
向下 t = 向下In t []
```

```
向下进入 :: BT a -> [BTStep a] -> BT (BTZ a)
向下进入 (a :& ft) ads = BTZ { 此处 = a, ctxt = DBT { 下方 = ft, 上方 = ads } } :& 进一步进入 a ft ads
```

注意，`a` 被替换为一个以 `a` 为焦点的拉链。子树由另一个辅助函数处理。

```

furtherIn :: a -> TF (BT a) -> [BTStep a] -> TF (BT (BTZ a)) furtherIn a Leaf ads = Leaf
furtherIn a (Fork (l, r)) ads = Fork ( downIn l ((a, Left ((), r)) : ads) , downIn r ((a, Right (
l, ())) : ads) )

```

可以看到，`furtherIn` 保留了树结构，但在访问子树时会适当地扩展子树上下文。

让我们再确认一下。

```

向下 aTree = BTZ { 这里 = 0, 上下文 = DBT { 下方 = Fork (1 :& Fork (2 :& Leaf, 3 :& Leaf), 4 :& Leaf), 上方 = [] } :&
Fork ( BTZ { 这里 = 1, 上下文 = DBT { 下方 = Fork (2 :& Leaf, 3 :& Leaf), 上方 = [(0, Left ((), 4 :& Leaf))] } :& Fork ( B
TZ { 这里 = 2, 上下文 = DBT { 下方 = Leaf, 上方 = [(1, Left ((), 3 :& Leaf)), (0, Left ((), 4 :& Leaf))] } :& Leaf , BTZ { 这
里 = 3, 上下文 = DBT { 下方 = Leaf, 上方 = [(1, Right (2 :& Leaf, ())), (0, Left ((), 4 :& Leaf))] } :& Leaf ) , BTZ { 这里 =
4, 上下文 = DBT { 下方 = Leaf, 上方 = [(0, Right (1 :& Fork (2 :& Leaf, 3 :& Leaf), ())) ] } :& Leaf)

```

看到了吗？每个元素都由其完整的上下文进行修饰，而不仅仅是它下面的那棵树。

二叉树拉链构成一个余单子

既然我们可以用它们的上下文来装饰元素，那么让我们构建 `Comonad` 实例。和之前一样……

实例 `Comonad BTZ`，其中提取 = 这里

... `extract` 告诉我们当前聚焦的元素，我们可以利用现有的机制进一步深入树结构，但我们需要构建新的工具来探索向外扩展的方式。

```

重复 z@(BTZ { 这里 = a, ctxt = DBT { 下方 = ft, 上方 = ads } }) = BTZ { 这里 = z, ctxt = DBT { 下方 = furt
herIn a ft ads -- 移动到 a 下面某处, 上方 = go_a (a :& ft) ads -- go 上方 a } } 其中

```

要向外移动，与列表一样，我们必须沿着路径向根节点回退。与列表一样，路径上的每一步都是我们可以访问的位置。

```

go_a t [] = [] go_a t (ad : ads) = go_ad t ad ads : go_a (plugBTinBT t ad) ads go_ad t (a, d) ads = BTZ { 这里 = a, ctxt = DB
T { 下面 = plugF t d, 上面 = ads } } -- 访问这里, go_d t a d ads -- 尝试其他 s

```

)

与列表不同，在那条路径上还有可供探索的替代分支。凡是该路径存储了一个尚未访问的子树的地方，我们都必须用 *their* 上下文来修饰 *its* 元素。

```
go_d t a (Left ((), r)) ads = Left ((), downIn r ((a, Right (t, ())) : ads))
go_d t a (Right (l, ())) ads = Right (downIn l (
(a, Left ((), t)) : ads), ())
```

所以现在已经解释了如何从任意元素位置重新聚焦到另一个位置。让我们看看。这里我们正在访问 1:

重复 (BTZ {这里 = 1, 上下文 = DBT { 下方 = 分叉 (2 :& 叶, 3 :& 叶), 上方 = [(0, 左 ((), 4 :& 叶))]}) = BTZ {这里 = BTZ {这里 = 1, 上下文 = DBT { 下方 = 分叉 (2 :& 叶, 3 :& 叶), 上方 = [(0, 左 ((), 4 :& 叶))]}) , 上下文 = DBT { 下方 = 分叉 (BTZ {这里 = 2, 上下文 = DBT { 下方 = 叶, 上方 = [(1, 左 ((), 3 :& 叶)), (0, 左 ((), 4 :& 叶))]}) :&

, BTZ {这里 = 3, 上下文 = DBT { 下方 = Leaf, 上方 = [(1, Right (2 :& Leaf, ())), (0, Left ((), 4 :& Leaf))]}) :&

, 上方 = [(BTZ {此处 = 0, 上下文 = DBT { 下方 = 分叉 (1 :& 分叉 (2 :& 叶, 3 :& 叶), 4 :& 叶), 上方 = []}] , 左 ((), BTZ {此处 = 4, 上下文 = DBT { 下方 = 叶, 上方 = [(0, 右 (1 :& 分叉 (2 :& 叶, 3 :& 叶), (0))]})

)]])

通过在一小部分数据上测试共单子法则，让我们检查：

```
fmap (\ z -> 提取 (复制 z) == z) (下 aTree) = 真 :& 分叉 (真 :& 分叉 (真 :& 叶, 真 :& 叶), 真 :& 叶)
fmap (\ z -> fmap 提取 (复制 z) == z) (下 aTree) = 真 :& 分叉 (真 :& 分叉 (真 :& 叶, 真 :& 叶), 真 :& 叶)
fmap (\ z -> fmap 复制 (复制 z) == 复制 (复制 z)) (下 aTree) = 真 :& 分叉 (真 :& 分叉 (真 :& 叶, 真 :& 叶), 真 :& 叶)
```

## 6.5 Data.Void 中的 absurd 函数有什么用？

考虑这种由自由变量参数化的lambda项表示法。（参见Bellegarde和Hook 1994年，Bird和Paterson 1999年，Altenkirch和Reus 1999年的论文。）

数据 Tm a = Var a | Tm a :\$ Tm a | Lam (Tm (Maybe a))

你当然可以把它做成一个 Functor，用来刻画重命名的概念，以及一个 Monad，用来刻画替换的概念。

Functor Tm 的实例，其中

```
fmap rho (Var a) = Var (rho a)
fmap rho (f :$ s) = fmap rho f :$ fmap rho s
```

```
fmap rho (Lam t) = Lam (fmap (fmap rho) t)
```

实例 Monad Tm, 其中 return =  
Var

```
变量 a >>= 信号 = 信号 a (f :$ s) >>= 信号 = (f >>= 信号) :$ (s >>= 信号) Lam t >>= 信号 = Lam  
(t >>= 也许 (Var Nothing) (fmap Just . sig))
```

现在考虑 *closed* 项: 它们是 Tm Void 的元素。你应该能够将封闭项嵌入到具有任意自由变量的项中。如何?

```
fmap absurd :: Tm Void -> Tm a
```

当然, 关键在于这个函数将遍历项, 但实际上什么也不做。不过, 它比 `unsafeCoerce` 更诚实一些。这也是为什么 `vacuous` 被添加到 `Data.Void` 中的原因。

Or 编写一个求值器。以下是包含自由变量的值 在 b。

数据 Val b = b :\$ [Val b] -- 一个卡住的应用 | 对于所有 a. LV (a -> Val b) (Tm (Maybe a)) -- 我们有一个不完整的环境

我刚刚将λ表达式表示为闭包。评估器通过一个环境来参数化, 该环境将a中的自由变量映射到b上的值。

```
eval :: (a -> Val b) -> Tm a -> Val b  
eval g (Var a) = g a  
eval g (f :$ s) = eval g f $ eval g s  
eval g (where (b :$ vs) v = b :$ vs ++ [v]) -- 卡住的应用变得更长  
LV g t $ v = eval (maybe v g) t -- 被应用的 lambda 解除卡住状态  
eval g (Lam t) = LV g t
```

你猜对了。在任意目标处对一个封闭项进行求值

```
评估荒谬 :: Tm Void -> Val b
```

更一般地说, `Void` 很少单独使用, 但当你想以一种表明某种不可能性的方式来实例化类型参数时, 它会很方便 (例如这里, 在封闭项中使用自由变量)。这些参数化类型通常带有高阶函数, 用于将对参数的操作提升为对整个类型的操作 (例如这里的 `fmap`、`>>=`、`eval`)。因此, 你可以将 `absurd` 作为作用于 `Void` 的通用操作来传递。

再举一个例子, 设想使用 `Either e v` 来捕获那些期望返回一个 `v`、但可能会抛出类型为 `e` 的异常的计算。你可以用这种方法来以统一的方式记录不良行为的风险。对于在这种设定下完全表现良好的计算, 可以令 `e` 为 `Void`, 然后使用

```
要么荒谬 id :: 要么 Void v -> v
```

安全运行或

```
要么荒谬的右侧 :: 要么虚空 v -> 要么 e v
```

将安全组件嵌入不安全的世界。

哦, 还有最后一项大动作, 处理一个“不能发生”的情况。它出现在通用拉链结构中, 出现在光标无法到达的每个地方。

```
class Differentiable f where type D f :: * -> * -- 带有一个洞的 f
plug :: (D f x, x) -> f x -- 将一个子节点塞入洞中
newtype K a x = K a -- 没有子节点, 只有一个标签
newtype I x = I x -- 一个子节点
data (f :+: g) x = L (f x) -- 选择
| R (g x)
data (f :*: g) x = f x :&: g x -- 配对
instance Differentiable (K a) where type D (K a) = K Void -- 没有子节点, 因此无法制造一个洞
plug (K v, x) = absurd v -- 无法重新发明标签, 因此拒绝这个洞!
```

我决定不删除其余部分, 尽管它并不完全相关。

实例 可微分 I 其中类型  $D I = K ()$  插入  $(K (), x) = I x$

实例  $(Differentiable f, Differentiable g) \Rightarrow Differentiable (f :+: g)$  其中类型  $D (f :+: g) = D f :+: D g$   
 $plug (L df, x) = L (plug (df, x))$   
 $plug (R dg, x) = R (plug (dg, x))$

实例 (可微 f, 可微 g)  $\Rightarrow$  可微  $(f :*: g)$  其中类型  $D (f :*: g) = (D f :*: g) :+: (f :*: D g)$   
 插件  $(L (df :&: g), x) =$  插件  $(df, x) :&: g$   
 插件  $(R (f :&: dg), x) = f :&:$  插件  $(dg, x)$

其实, 也许这很相关。如果你有冒险精神, 这篇未完成的文章展示了如何使用 Void 来压缩包含自由变量的项的表示。

```
data Term f x = Var x | Con (f (Term f x)) -- 又一次的自由单子
```

在任何由一个可微且可遍历的函子 f 自由生成的语法中。我们使用  $Term f Void$  来表示不含自由变量的区域, 并使用  $[D f (Term f Void)]$  来表示 *tubes* 穿行通过同样不含自由变量的区域, 通向一个孤立的自由变量, 或通向通往两个或更多自由变量的路径中的一个汇合点。总有一天得把那篇文章写完。

对于一个没有任何值 (或者至少, 没有什么适合在体面场合谈论的值) 的类型来说, Void 却异常有用。而你对它的用法则荒诞不经。

## 6.6 为 n 维网格编写 cojoin 或 cobind

Jagger/Richards: 你不可能总是得到你想要的, 但如果你有时努力尝试, 你或许会发现你得到了你需要的。

### 6.6.1 列表中的光标

让我使用 `snoc` 列表和 `cons` 列表重建你结构的组件, 以保持空间属性的清晰。我定义

数据  $\text{Bwd } x = \text{B0} \mid \text{Bwd } x :< x$  派生 (Functor, Foldable, Traversable, Show) 数据  $\text{Fwd } x = \text{F0} \mid x :> \text{Fwd } x$  派生 (Functor, Foldable, Traversable, Show)  $\text{infixl } 5 :<$

$\text{infixr } 5 :> \text{da}$

$\text{ta Cursor } x = \text{Cur } (\text{Bwd } x) \ x \ (\text{Fwd } x) \text{ deriving (Functor, Foldable, Traversable, Show)}$

让我们来看看 *comonad* (余单子)

类 函子  $f \Rightarrow$  共单态  $f$  其中

$\text{counit} :: f \ x \rightarrow x \ \text{cojoin} :: f \ x \rightarrow f \ (f \ x)$

并且确保光标是共同态

实例 *Comonad Cursor*, 其中  $\text{counit } (\text{Cur } \_ \ x \ \_) = x \ \text{cojoin } c = \text{Cur } (\text{lefts } c) \ c \ (\text{rights } c)$  其中  $\text{lefts } (\text{Cur } \text{B0 } \_ \ \_) = \text{B0} \ \text{lefts } (\text{Cur } (xz :< x) \ y \ ys) = \text{lefts } c :< c$  其中  $c = \text{Cur } xz \ x \ (y :> ys) \ \text{rights } (\text{Cur } \_ \ \text{F0}) = \text{F0} \ \text{rights } (\text{Cur } xz \ x \ (y :> ys)) = c :> \text{rights } c$  其中  $c = \text{Cur } (xz :< x) \ y \ ys$

如果你对这类东西感兴趣, 你会注意到 *Cursor* 是 *InContext []* 的一种在空间上更令人愉悦的变体。

上下文中  $f \ x = (x, \partial f \ x)$

其中  $\partial$  对一个函子取形式导数, 从而给出其“一孔”上下文的概念。 *InContext f* 始终是一个 *Comonad*, 正如这个回答中提到的那样, 而我们这里得到的正是由微分结构所诱导的那个 *Comonad*: 其中 *counit* 提取焦点处的元素, 而 *cojoin* 为每个元素装饰上它自身的上下文, 从而有效地给你一个充满重新聚焦游标的上下文, 并且在其焦点处有一个未移动的游标。让我们来看一个例子。

$> \text{共连接 } (\text{Cur } (\text{B0} :< 1) \ 2 \ (3 :> 4 :> \text{F0})) \ \text{Cur } (\text{B0} :< \text{Cur } \text{B0} \ 1 \ (2 :> 3 :> 4 :> \text{F0})) \ (\text{Cur } (\text{B0} :< 1) \ 2 \ (3 :> 4 :> \text{F0})) \ (\text{Cur } (\text{B0} :< 1 :< 2) \ 3 \ (4 :> \text{F0}) :> \text{Cur } (\text{B0} :< 1 :< 2 :< 3) \ 4 \ \text{F0} :> \text{F0})$

看到了吗? 聚焦的 2 已被修饰为光标在 2; 左边是光标在 1 的列表; 右边是光标在 3 和光标在 4 的列表。

### 6.6.2 组合游标, 转置游标?

现在, 你所询问要成为一个 *Comonad* 的结构是 *Cursor* 的  $n$  重复合。让我们有

新类型  $(::) \ f \ g \ x = C \ \{\text{unC} :: f \ (g \ x)\}$  派生 *Show*

要使余单子  $f$  和  $g$  复合, 它们的余单位可以很好地复合, 但你需要一个“分配律”。

转置  $:: f \ (g \ x) \rightarrow g \ (f \ x)$

因此你可以像这样构建复合 *cojoin*

```
f (g x) -(fmap cojoin)-> f (g (g
x)) -cojoin-> f (f (g (g x))) -(fm
ap transpose)-> f (g (f (g x)))
```

W帽子法律应该转置满足吗？可能是某种像

```
counit . 转置 = fmap counit cojoin . 转置 = fmap 转置 . 转置 . fmap cojoin
```

或采取任何措施确保将 f 和 g 的某个序列从一种顺序移到另一种顺序的两种方法得到相同的结果。

我们可以为 Cursor 自身定义一个转置吗？获得某种转置的廉价方法之一是注意到 Bwd 和 Fwd 是 *zippily* 应用型的，因此 Cursor 也是如此。

实例 Applicative Bwd 其中  $\text{pure } x = \text{pure } x :< x \text{ (fz :< f) } <*> \text{ (sz :< s) } = (\text{fz } <*> \text{ sz}) :< f \text{ s } \_ <*> \_ = \text{B0}$

实例 Applicative Fwd 其中

```
纯 x = x :> 纯 x (f :> fs) <*> (s :> ss) = f s :> (fs <*> ss) _
<*> _ = F0
```

实例 Applicative 游标，其中

```
纯 x = Cur (纯 x) x (纯 x) Cur fz f fs <*> Cur sz s ss = Cur (fz <*> sz) (f s) (fs <*> ss)
```

在这里你应该开始察觉到问题。形状不匹配会导致 *truncation*，这将破坏自转置是自逆的显然理想属性。任何形式的不规则性都无法存活。我们确实得到了一个转置算子：sequenceA，对于完全规则的数据，一切都是光明和美好的。

> 常规矩阵游标

```
Cur (B0 :< Cur (B0 :< 1) 2 (3 :> F0)) (Cur (B0 :< 4) 5 (6 :> F0)
) (Cur (B0 :< 7) 8 (9 :> F0) :> F0)
```

> 序列A 常规矩阵游标

```
Cur (B0 :< Cur (B0 :< 1) 4 (7 :> F0)) (Cur (B0 :< 2) 5 (8 :> F0)
) (Cur (B0 :< 3) 6 (9 :> F0) :> F0)
```

但即使我只是将其中一个内游标移出对齐（更不用说让大小变得不整齐），事情也会出错。

> 不规则矩阵光标

```
Cur (B0 :< Cur ((B0 :< 1) :< 2) 3 F0) (Cur (B0 :< 4) 5 (6 :> F0)
) (Cur (B0 :< 7) 8 (9 :> F0) :> F0)
```

> 序列A 不规则矩阵游标

```
Cur (B0 :< Cur (B0 :< 2) 4 (7 :> F0)
      (Cur (B0 :< 3) 5 (8 :> F0)
      F0
```

当你有一个外部光标位置和多个内部光标位置时，没有任何转置会表现得很好。自组成光标允许内部结构相对于彼此呈不规则排列，因此没有转置，也没有合并。你可以，我也做了，定义

我实例 (Comonad f, Traversable f, Comonad g, Applicative g) => Comonad (f :: g) 其中 counit = counit . counit . unC cojoin = C . fmap (fmap C . sequenceA) . cojoin . fmap cojoin . unC

但我们有责任确保保持内部结构的规则性。如果你愿意接受这个负担，那么你可以进行迭代，因为 Applicative 和 Traversable 在合成下是封闭的。这里是一些片段和部分

实例 (函子 f, 函子 g) => 函子 (f :: g) 其中 fmap h (C fgx) = C (fmap (fmap h) fgx)

实例 (Applicative f, Applicative g) => Applicative (f :: g) 其中 纯 = C . 纯 . 纯 C f <\*> C s = C (纯 (<\*>) <\*> f <\*> s)

instance (Functor f, Foldable f, Foldable g) => Foldable (f :: g) where fold = fold . fmap fold . unC

实例 (Traversable f, Traversable g) => Traversable (f :: g) 其中 traverse h (C fgx) = C <\$> traverse (traverse h) fgx

编辑：为完整起见，这是在一切正常时它的表现，

> 联接 (C 常规矩阵光标) C {unC = 光标 (B0 :< 光标 (B0 :< C {unC = 光标 B0 (光标 B0 1 (2 :> (3 :> F0) )) (光标 B0 4 (5 :> (6 :> F0) )) :> (光标 B0 7 (8

(C {unC = Cur B0 (Cur (B0 :< 1) 2 (3 :> F0)) (Cur (B0 :< 4) 5 (6 :> F0) :> (Cur (B0 :< (C {unC = Cur B0 (Cur ((B0 :< 1) :< 2) 3 F0) (Cur ((B0 :< 4) :< 5) 6 F0 :> (Cur ((B0 :< (Cur (B0 :< C {unC = Cur (B0 :< Cur B0 1 (2 :> (3 :> F0) )) (Cur B0 4 (5 :> (6 :> F0) )) (Cur B0 7 (

(C {unC = Cur (B0 :< Cur (B0 :< 1) 2 (3 :> F0)) (Cur (B0 :< 4) 5 (6 :> F0)) (Cur (B0 :< (C {unC = Cur (B0 :< Cur ((B0 :< 1) :< 2) 3 F0) (Cur ((B0 :< 4) :< 5) 6 F0) (Cur ((B0 :< (Cur (B0 :< C {unC = Cur ((B0 :< Cur B0 1 (2 :> (3 :> F0) )) :< Cur B0 4 (5 :> (6 :> F0) )) (Cur B0

(C {unC = Cur ((B0 :< Cur (B0 :< 1) 2 (3 :> F0)) :< Cur (B0 :< 4) 5 (6 :> F0)) (Cur (B0 :< (C {unC = Cur ((B0 :< Cur ((B0 :< 1) :< 2) 3 F0) :< Cur ((B0 :< 4) :< 5) 6 F0) (Cur ((B :> F0))

### 6.6.3 Hancock 的张量积

为了正则性，你需要比复合更强的东西。你需要能够把握“由形状完全相同的 g-结构构成的 f-结构”这一概念。这正是那位不可多得的 Peter Hancock 所称的“张量积”，我将其记作 f:><: g: 这里有一个“外层”的 f-形状，以及一个对所有内部 g-结构都相同的“内层” g-形状，因此转置很容易定义，而且总是自逆的。Hancock 的张量在 Haskell 中并不便于定义，但在依赖类型的设定中，很容易形式化一种具有这种张量的“容器”概念。

为了让你理解，考虑一个退化的容器概念

```
data (:<|) s p x = s :<| (p -> x)
```

我们说  $s$  是“形状”的类型， $p$  是“位置”的类型。一个值由选择一个形状和在每个位置存储一个  $x$  组成。在依赖的情况下，位置的类型可能依赖于形状的选择（例如，对于列表，形状是一个数字（长度），然后你有这么多个位置）。这些容器具有张量积。

$$(s :<| p) :>< (s' :<| p') = (s, s') :<| (p, p')$$

这就像一个广义的矩阵：一对形状给出维度，然后在每一对位置上都有一个元素。当类型  $p$  和  $p'$  依赖于  $s$  和  $s'$  中的值时，你也完全可以这样做，而这正是 Hancock 对容器的张量积的定义。

### 6.6.4 张量积的上下文

现在，正如你可能在高中学到的那样， $\partial (s :<| p) = (s, p) :<| (p-1)$ ，其中  $p-1$  是比  $p$  少一个元素的某种类型。像  $\partial (sx^p) = (sp)x^{(p-1)}$ . *You select one position (recording it in the shape) and delete it. The snag is that  $p-1$  is tricky to get your hands on without dependent types. But InContext selects a position, 并且不会删除它\*。*

`InContext (s :<| p) ~ = (s, p) :<| p`

这在依赖情况下同样适用，我们也欣然地获得

在上下文  $(f :>< g) \sim =$  在上下文  $f :><$  在上下文  $g$

现在我们知道 `InContext f` 总是一个余单子，这告诉我们 `InContext` 的张量积是余单子的，因为它们本身就是 `InContext`。也就是说，你在每个维度上选择一个位置（这会在整体中给出恰好一个位置），而此前我们是一个外部位置加上许多内部位置。用张量积取代复合之后，一切都运作得很顺畅。

### 6.6.5 纳皮尔函子

但是有一类 Functor，其张量积与复合是重合的。这些是满足  $f() \sim ()$  的 Functor  $f$ ：也就是说，无论如何都只有一种形状，因此在一开始就排除了组合中的参差不齐的值。这些 Functor 都与某个位置集合  $p$  的  $(p \rightarrow)$  同构，我们可以把  $p$  看作是 *logarithm*（为了得到  $f x$ ）， $x$  必须被提升到的指数。相应地，Hancock 以约翰·纳皮尔（John Napier）之名将这些称为 Naperian functor（纳皮尔幽灵出没在 Hancock 居住的爱丁堡一带）。

类 `Applicative f => Naperian f` 其中类型 `Log f` 项目 `:: f`  
`x -> Log f -> x 位置 :: f (Log f) --- 项目 位置 = id`

一个 Naperian 函子具有对数，诱导一个投影函数，将位置映射到那里找到的元素。Naperian 函子都是迅速的 `Applicative`，纯函数和 `<*>` 分别对应于投影的 `K` 和 `S` 组合子。还可以构造一个值，其中每个位置存储该位置的表示。你可能记得的对数法则愉快地浮现出来。

新类型 `Id x = Id {unId :: x}` 派生 `Sho`

`w`

实例 `Naperian Id` 其中类型 `Log Id`

`= () 项目 (Id x) () = x`

位置 = 身份 ()

新 类型  $(:*)$   $f\ g\ x = \text{Pr}\ (f\ x, g\ x)$  派生 显示

i 实例  $(\text{Naperian}\ f, \text{Naperian}\ g) \Rightarrow \text{Naperian}\ (f\ ::\ g)$  其中类型  $\text{Log}\ (f\ ::\ g) = \text{Either}\ (\text{Log}\ f)\ (\text{Log}\ g)$   
 $\text{project}\ (\text{Pr}\ (f\ x, g\ x))\ (\text{Left}\ p) = \text{project}\ f\ x\ p$   
 $\text{project}\ (\text{Pr}\ (f\ x, g\ x))\ (\text{Right}\ p) = \text{project}\ g\ x\ p$   
 $\text{positions} = \text{Pr}\ (\text{fmap}\ \text{Left}\ \text{positions}, \text{fmap}\ \text{Right}\ \text{positions})$

请注意，一个固定大小的数组（一个 *vector*）由  $(\text{Id}\ ::\ \text{Id}\ ::\ \dots\ ::\ \text{Id}\ ::\ \text{One})$  给出，其中 *One* 是常量单位函子，其对数为 *Void*。因此，一个数组是 *Naperian* 的。现在，我们还有

实例  $(\text{Naperian}\ f, \text{Naperian}\ g) \Rightarrow \text{Naperian}\ (f\ ::\ g)$  其中 类型  $\text{Log}\ (f\ ::\ g) = (\text{Log}\ f, \text{Log}\ g)$   
 $\text{投影}\ (\text{C}\ \text{fgx})\ (p, q) = \text{投影}\ (\text{投影}\ \text{fgx}\ p)\ q$  位置 =  $\text{C}\ \$\ \text{fmap}\ (\backslash p \rightarrow \text{fmap}\ (p, \text{位置}))\ \text{位置}$

这意味着多维数组是纳佩里安的。要构造一个适用于纳佩里安  $f$  的  $\text{InContext}\ f$  版本，只需指向一个位置！

数据聚焦  $f\ x = f\ x\ ::\ @\ \text{日志}\ f$

实例  $\text{Functor}\ f \Rightarrow \text{Functor}\ (\text{Focused}\ f)$  其中  $\text{fmap}\ h\ (f\ x\ ::\ @\ p) = \text{fmap}\ h\ f\ x\ ::\ @\ p$

实例  $\text{Naperian}\ f \Rightarrow \text{Comonad}\ (\text{Focused}\ f)$  其中  $\text{counit}\ (f\ x\ ::\ @\ p) = \text{project}\ f\ x\ p$   
 $\text{cojoin}\ (f\ x\ ::\ @\ p) = \text{fmap}\ (f\ x\ ::\ @)\ \text{positions}\ ::\ @\ p$

因此，特别地，一个聚焦的  $n$  维数组确实将是一个余单子。向量的复合是  $n$  个向量的张量积，因为向量是 *Naperian* 的。而聚焦的  $n$  维数组将是决定其各个维度的  $n$  个聚焦向量的  $n$  重张量积 *not the composition*。为了用拉链来表达这个余单子，我们需要将它们表示成一种能够构造张量积的形式。我将把这留作未来的练习。

## 6.7 拉链共单元，通用地

就像《飞天万能车》里的捕孩人用糖果和玩具把孩子们诱进囚禁一样，本科物理专业的招生者也喜欢拿肥皂泡和回旋镖来糊弄人；可当门眶当一声关上时，便成了：“好了，孩子们，该学习偏微分了！”我也是。别说我没警告过你。

这是另一个警告：以下代码需要 `{-# LANGUAGE KitchenSink #-}`，或者说

```
{-# LANGUAGE TypeFamilies, FlexibleContexts, TupleSections, GADTs, DataKinds, TypeOperators, FlexibleInstances, RankNTypes, ScopedTypeVariables, StandaloneDeriving, UndecidableInstances #-}
```

不分先后顺序。可微函子产生余单子拉链。那么，究竟什么是可微函子？

类 (Functor f, Functor (DF f)) => Diff1 f 其中类型  $DF\ f :: * \rightarrow * \text{ upF} ::$   
 $ZF\ f\ x \rightarrow f\ x \text{ downF} :: f\ x \rightarrow f\ (ZF\ f\ x) \text{ aroundF} :: ZF\ f\ x \rightarrow ZF\ f\ (ZF\ f\ x)$

数据  $ZF\ f\ x = (\prec\prec) \{cxF :: DF\ f\ x, elF :: x\}$

它是一个具有导数的函子，导数也是一个函子。导数表示一个 *element* 的一孔上下文。拉链类型  $ZF\ f\ x$  表示一孔上下文和孔中元素的配对。

Diff1 的操作描述了我们可以在拉链结构上进行的各种导航方式（不涉及“向左”和“向右”的概念，关于这一点请参见我关于 Clowns and Jokers 的论文）。我们可以“向上”，通过将元素插回其孔位来重新组装结构。我们可以“向下”，找出在给定结构中访问某个元素的所有方式：我们用其上下文来装饰每一个元素。我们还可以“绕行”，即取一个已有的拉链，并用其上下文来装饰每一个元素，从而找到所有重新聚焦的方式（以及如何保持我们当前的焦点）。

现在，aroundF 的类型可能会让你们中的一些人想起

类 Functor c => Comonad c, 其中  $extract :: c\ x \rightarrow x \text{ duplicate} :: c\ x \rightarrow c\ (c\ x)$

你提醒得对！我们已经，一跳一蹦，

实例 Diff1 f => 函子 (ZF f) 其中  $fmap\ f\ (df\ \prec\prec\ x) = fmap\ f\ df\ \prec\prec\ f\ x$

实例 Diff1 f => Comonad (ZF f) 其中 提取 = elF 复制 = aroundF

并且我们坚持认为

提取 . 复制 == 恒等 映射 提取 . 复制 == 恒等 复制 . 复制 == 映射 复制 . 复制

我们也需要这个

$fmap\ extract\ (downF\ xs) == xs \text{ -- downF 为对应位置的元素添加装饰}$   
 $fmap\ upF\ (downF\ xs) = fmap\ (const\ xs)\ xs \text{ -- downF 提供正确的上下文}$

多项式函子是可微的。常值函子是可微的。

数据  $KF\ a\ x = KF\ a$  实例函子 (KF a) 其中  
 $fmap\ f\ (KF\ a) = KF\ a$

实例 Diff1 (KF a) 其中 类型  $DF\ (KF\ a) = KF\ \text{空}$   
 $upF\ (KF\ w\ \prec\prec\ \_) = \text{荒谬}\ w \text{ downF}\ (KF\ a) = KF\ a$   
 $aroundF\ (KF\ w\ \prec\prec\ \_) = \text{荒谬}\ w$

没有地方放置元素，因此无法形成上下文。没有可以向上F或向下F的位置，我们很容易发现没有任何方式可以向下F。

恒等函子是可微的。

数据  $\text{IF } x = \text{IF } x$  实例  $\text{Functor IF}$  其中  $\text{fmap } f (\text{IF } x) = \text{IF } (f x)$   
 x) 实例  $\text{Diff1 IF}$  其中 类型  $\text{DF IF} = \text{KF } () \text{ upF } (\text{KF } () :<-: x) = \text{IF } x \text{ downF } (\text{IF } x) = \text{IF } (\text{KF } () :<-: x) \text{ aroundF } z @ (\text{KF } () :<-: x) = \text{KF } () :<-: z$

在一个平凡的上下文中只有一个元素， $\text{downF}$  找到它， $\text{upF}$  将其重新打包，而  $\text{aroundF}$  只能原地不动。

求和保持可微性。

数据  $(f :+ : g) x = \text{LF } (f x) \mid \text{RF } (g x)$  实例  $(\text{Functor } f, \text{Functor } g) \Rightarrow \text{Functor } (f :+ : g)$   
 其中  $\text{fmap } h (\text{LF } f) = \text{LF } (\text{fmap } h f) \text{ fmap } h (\text{RF } g) = \text{RF } (\text{fmap } h g)$

实例  $(\text{Diff1 } f, \text{Diff1 } g) \Rightarrow \text{Diff1 } (f :+ : g)$  其中类型  $\text{DF } (f :+ : g) = \text{DF } f :+ : \text{DF } g$   
 $\text{F } g \text{ upF } (\text{LF } f' :<-: x) = \text{LF } (\text{upF } (f' :<-: x)) \text{ upF } (\text{RF } g' :<-: x) = \text{RF } (\text{upF } (g' :<-: x))$

其他细节则稍微复杂一些。要下F，我们必须在标记组件内部下F，然后修复结果的拉链以显示上下文中的标签。

```
downF (LF f) = LF (fmap (\ (f' :<-: x) -> LF f' :<-: x) (downF f))
downF (RF g) = RF (fmap (\ (g' :<-: x) -> RF g' :<-: x) (downF g))
```

为了实现  $\text{aroundF}$ ，我们先去掉标签，弄清楚如何对未加标签的对象进行  $\text{around}$ ，然后在所有得到的拉链中恢复该标签。当前聚焦的元素  $x$  被其完整的拉链  $z$  所替换。

```
围绕F z@(LF f' :<-: (x :: x)) = LF (fmap (\ (f' :<-: x) -> LF f' :<-: x) . cxF $ 围绕F (f' :<-: x :: ZF f x)) :<-: z 围绕F z@(RF g' :<-: (x :: x)) = RF (fmap (\ (g' :<-: x) -> RF g' :<-: x) . cxF $ 围绕F (g' :<-: x :: ZF g x)) :<-: z
```

请注意，我不得不使用 `ScopedTypeVariables` 来消除对  $\text{aroundF}$  的递归调用的歧义。作为一种类型函数， $\text{DF}$  不是单射，因此  $f' :: \text{D } f x$  不足以强制  $f' :<-: x :: \text{Z } f x$ 。

乘积保持可微性。

数据  $(f :* : g) x = f x :* : g x$  实例  $(\text{Functor } f, \text{Functor } g) \Rightarrow \text{Functor } (f :* : g)$  其中  $\text{fmap } h (f :* : g) = \text{fmap } h f :* : \text{fmap } h g$

为了集中关注一对元素中的一个，你要么关注左侧，保持右侧不变，要么反之亦然。莱布尼茨著名的乘积法则对应于一种简单的空间直觉！

实例  $(\text{Diff1 } f, \text{Diff1 } g) \Rightarrow \text{Diff1 } (f : * : g)$  其中类型  $\text{DF } (f : * : g) = (\text{DF } f : * : g) : + : (f : * : \text{DF } g) \text{ upF } (\text{LF } (f' : * : g) : < - : x) = \text{upF } (f' : < - : x) : * : g \text{ upF } (\text{RF } (f : * : g') : < - : x) = f : * : \text{upF } (g' : < - : x)$

现在，`downF` 的工作方式类似于它在求和时的工作方式，唯一不同的是我们不仅需要用个标签来修正拉链上下文（以显示我们走了哪条路径），还需要修正另一个未触及的组件。

$$\text{downF } (f : * : g) = \text{fmap } (\lambda (f' : < - : x) \rightarrow \text{LF } (f' : * : g) : < - : x) (\text{downF } f) : * : \text{fmap } (\lambda (g' : < - : x) \rightarrow \text{RF } (f : * : g') : < - : x) (\text{downF } g)$$

但`aroundF`是一个充满欢笑的大袋子。无论我们目前访问的是哪一方，我们都有两个选择：

1. 在那一侧移动F。
2. 从那一侧向上移动F，再向下F移动到另一侧。

每种情况都需要我们利用子结构的操作，然后修正上下文。

围绕F  $z @ (\text{LF } (f' : * : g) : < - : (x :: x)) = \text{LF } (\text{fmap } (\lambda (f' : < - : x) \rightarrow \text{LF } (f' : * : g) : < - : x) (\text{cxF } \$ \text{围绕F } (f' : < - : x :: \text{ZF } f x)) : * : \text{fmap } (\lambda (g' : < - : x) \rightarrow \text{RF } (f : * : g') : < - : x) (\text{downF } g)) : < - : z$  其中  $f = \text{upF } (f' : < - : x)$  围绕F  $z @ (\text{RF } (f : * : g') : < - : (x :: x)) = \text{RF } (\text{fmap } (\lambda (f' : < - : x) \rightarrow \text{LF } (f' : * : g) : < - : x) (\text{downF } f) : * : \text{fmap } (\lambda (g' : < - : x) \rightarrow \text{RF } (f : * : g') : < - : x) (\text{cxF } \$ \text{围绕F } (g' : < - : x :: \text{ZF } g x)) : < - : z$  其中  $g = \text{upF } (g' : < - : x)$

呼！这些多项式都是可微的，因此给我们提供了共单准范畴。

嗯。这有点抽象。所以我在我能做的地方都加上了派生 `Show`，并且加入了

派生实例  $(\text{Show } (\text{DF } f x), \text{Show } x) \Rightarrow \text{Show } (\text{ZF } f x)$

允许了以下互动（手动整理过）

```
> downF (IF 1 : * : IF 2) IF (LF (KF () : * : IF 2) : < - : 1) : * : IF (RF (IF 1 : * : KF ()) : < - : 2)
```

```
> fmap aroundF 它
```

```
如果 (LF (KF () : * : 如果 (RF (如果 1 : * : KF ()) : < - : 2)) : < - : (LF (KF () : * : 如果 2) : < - : 1)) : * :
```

```
如果 (RF (如果 (LF (KF () : * : 如果 2) : < - : 1) : * : KF ()) : < - : (RF (如果 1 : * : KF ()) : < - : 2))
```

练习 证明可微函数的复合是可微的，使用 *chain rule*。

太棒了！我们现在可以回家了吗？当然不行。我们还没有区分任何 *recursive* 结构。

从双函子构造递归函子

双函子，正如现有关于数据类型泛型编程的文献（参见Patrik Jansson和Johan Jeuring的研究，或者Jeremy Gibbons的精彩讲义）详细解释的那样，是一个具有两个参数的类型构造器，对应于两种子结构。我们应该能够“映射”这两者。

类 Bifunctor b 其中

```
bimap :: (x -> x') -> (y -> y') -> b x y -> b x' y'
```

我们可以使用双函子来给出递归容器的节点结构。每个节点都有 *subnodes* 和 *elements*。它们可以只是两种子结构。

```
data Mu b y = In (b (Mu b y) y)
```

看到了吗？我们在 b 的第一个参数中“打结递归”，并将参数 y 保持在第二个参数中。因此，我们一次性得到

实例 Bifunctor b => Functor (Mu b), 其中 fmap f (In b) = In (bimap f f b)

为此，我们需要一组 Bifunctor 实例。

Bifunctor Kit 常量是双函子性的。

新类型 K a x y = K a

实例 Bifunctor (K a) 在其中 bimap f g (K a) = K a

你可以看出我先写了这一部分，因为标识符更短，但这反而是好事，因为代码更长。

变量是双函子性的。

我们需要对应于一个参数或另一个参数的双重函子，因此我创建了一个数据类型来区分它们，然后定义了一个合适的 GADT。

数据 Var = X | Y

数据 V :: Var -> \* -> \* -> \* 其中 XX :: x -> V X

x y YY :: y -> V Y x y

这使得 V X x y 成为 x 的副本，V Y x y 成为 y 的副本。因此

实例 Bifunctor (V v) 其中 bimap f g (XX x)

= XX (f x) bimap f g (YY y) = YY (g y)

双函子的和与积仍然是双函子

数据 (:+:.) f g x y = L (f x y) | R (g x y) 导出 Show

实例 (Bifunctor b, Bifunctor c) => Bifunctor (b :+: c) 其中 bimap f g (L b) = L (bimap f g b) bimap f g (R b) = R (bimap f g b)

数据 (:\*\*.) f g x y = f x y \*\*: g x y 导出 Show

实例 (Bifunctor b, Bifunctor c) => Bifunctor (b \*\*: c) 其中 bimap f g (b \*\*: c) = bimap f g b \*\*: bimap f g c

到目前为止，这些都只是套话，但现在我们可以定义例如

列表 = Mu (K () :++: (V Y :\*: V X))

B在 = Mu (V Y :\*: (K () :++: (V X :\*: V X

抱歉，您的输入似乎不完整。能否提供更多上下文或文本以便

如果你想将这些类型用于实际数据，而不是像乔治·修拉的点彩派传统那样盲目使用，请使用 *pattern synonyms*。

但是拉链呢？我们如何证明 Mu b 是可微的？我们需要证明 b 在 *both* 个变量下是可微的。铛！是时候学习偏微分了。

双函子的偏导数

由于我们有两个变量，我们有时需要能够共同讨论它们，有时又需要单独讨论它们。我们将需要单例族：

数据 Vary :: Var -> \* 其中 VX :: Vary

X VY :: Vary Y

现在我们可以说明双函子在每个变量上具有偏导数的含义，并给出相应的拉链概念。

类 (Bifunctor b, Bifunctor (D b X), Bifunctor (D b Y)) => Diff2 b 其中类型 D b (v :: Var) :: \* -> \* -> \*

向上 :: 变化 v -> Z b v x y -> b x y 向下 :: b x y -> b (Z b X x y) (Z b Y x y) 四周 :: 变化 v -> Z b v x y -> Z b v (Z b X x y) (Z b Y x y)

数据 Z b v x y = (:<-) {cxZ :: D b v x y, elZ :: V v x y}

此 D 操作需要知道目标变量是哪一个。对应的拉链 Z b v 告诉我们哪个变量 v 必须聚焦。当我们“装饰上下文”时，我们必须用 X-上下文装饰 x-元素，并用 Y-上下文装饰 y-元素。但除此之外，故事还是一样。

我们有两个剩余的任务：首先，证明我们的二元函数套件是可微的；其次，证明 Diff2 b 允许我们建立 Diff1 (Mu b)。

区分双函子工具包

我担心这一部分更像是琐碎的，而不是启发性的。随意跳过。

常数与之前相同。

实例 Diff2 (K a) 其中类型 D (K a) v = K Void

上 \_ (K q :<- \_) = 荒谬 q 下 (K a) = K a 周围 \_ (

K q :<- \_) = 荒谬 q

这次，人生太短暂，无法发展类型级别的克罗内克δ理论，所以我只是将变量分别处理。

实例 Diff2 (V X) 其中 类型 D (V X) X = K () 类型 D (V X) Y = K

Void 向上 VX (K () :<- XX x) = XX x 向上 VY (K q :<- \_) = 荒谬

q 向下 (XX x) = XX (K () :<- XX x) 周围 VX z@(K () :<- XX x) =

K () :<- XX z 周围 VY (K q :<- \_) = 荒谬 q

实例 Diff2 (V Y) 在

类型  $D(V\ Y)\ X = K$  空 类型  $D(V\ Y)\ Y = K()$  上  $VX(K\ q :<- \_)$   
 $) =$  荒谬  $q$  上  $VY(K() :<- YY\ y) = YY\ y$  下  $(YY\ y) = YY(K())$   
 $:<- YY\ y)$  环绕  $VX(K\ q :<- \_) =$  荒谬  $q$  环绕  $VY\ z@(K()) :<- Y$   
 $Y\ y) = K() :<- YY\ z$

对于结构性情况，我发现引入一个辅助函数很有用，它使我能够统一地处理变量。

$vV ::$  变化  $v \rightarrow Z\ b\ v\ x\ y \rightarrow V\ v\ (Z\ b\ X\ x\ y)\ (Z\ b\ Y\ x\ y)\ vV\ VX\ z = XX\ z\ vV\ VY\ z$   
 $= VY\ z$

然后我制作了一些小工具，以便促进我们在向下和绕行时所需要的那种“重新标记”。（当然，我是在工作过程中才看清自己需要哪些工具的。）

$zimap :: (Bifunctor\ c) \Rightarrow$  (对所有  $v$ .  $Vary\ v \rightarrow D\ b\ v\ x\ y \rightarrow D\ b' \ v\ x\ y \rightarrow c\ (Z\ b\ X\ x\ y)\ (Z\ b\ Y\ x\ y) \rightarrow c\ (Z\ b' \ X\ x\ y)\ (Z\ b' \ Y\ x\ y)$ )

$zimap\ f = bimap$   
 $(\backslash\ (d :<- XX\ x) \rightarrow f\ VX\ d :<- XX\ x)\ (\backslash\ (d :<- YY\ y)$   
 $\rightarrow f\ VY\ d :<- YY\ y)$

$dzimap :: (Bifunctor\ (D\ c\ X), Bifunctor\ (D\ c\ Y)) \Rightarrow$  (forall  $v$ .  $Vary\ v \rightarrow D\ b\ v\ x\ y \rightarrow D\ b' \ v\ x\ y \rightarrow Vary\ v \rightarrow Z\ c\ v\ (Z\ b\ X\ x\ y)\ (Z\ b\ Y\ x\ y) \rightarrow D\ c\ v\ (Z\ b' \ X\ x\ y)\ (Z\ b' \ Y\ x\ y)\ dzimap\ f\ VX\ (d :<- \_) = bimap\ (\backslash\ (d :<- XX\ x) \rightarrow f\ VX\ d :<- XX\ x)\ (\backslash\ (d :<- YY\ y) \rightarrow f\ VY\ d :<- YY\ y)\ d\ dzimap\ f\ VY\ (d :<- \_) = bimap\ (\backslash\ (d :<- XX\ x) \rightarrow f\ VX\ d :<- XX\ x)\ (\backslash\ (d :<- YY\ y) \rightarrow f\ VY\ d :<- YY\ y)\ d$

有了这套准备就绪的东西，我们就可以把细节逐一敲定了。求和很简单。

实例  $(Diff2\ b, Diff2\ c) \Rightarrow Diff2\ (b :++:\ c)$  其中类型  $D(b :++:\ c)\ v = D\ b\ v :++:\ D\ c\ v\ up\ v\ (L\ b' :<- vv) = L\ (up\ v\ (b' :<- vv))\ down\ (L\ b) = L\ (zimap\ (const\ L)\ (down\ b))\ down\ (R\ c) = R\ (zimap\ (const\ R)\ (down\ c))\ around\ v\ z@(L\ b' :<- vv :: Z\ (b :++:\ c)\ v\ x\ y) = L\ (dzimap\ (const\ L)\ v\ ba) :<- vV\ v\ z$  其中  $ba = around\ v\ (b' :<- vv :: Z\ b\ v\ x\ y)\ around\ v\ z@(R\ c' :<- vv :: Z\ (b :++:\ c)\ v\ x\ y) = R\ (dzimap\ (const\ R)\ v\ ca) :<- vV\ v\ z$  其中  $ca = around\ v\ (c' :<- vv :: Z\ c\ v\ x\ y)$

产品需要艰苦的工作，这就是为什么我是一名数学家而不是工程师。

实例  $(Diff2\ b, Diff2\ c) \Rightarrow Diff2\ (b :**:\ c)$  其中 类型  $D(b :**:\ c)\ v = (D\ b\ v :**:\ c) :++:\ (b :**:\ D\ c\ v)\ up\ v\ (L\ (b' :**:\ c) :<- vv) = up\ v\ (b' :<- vv) :**:\ c$

上  $v (R (b : ** : c') :<- vv) = b : ** : \text{上 } v (c' :<- vv)$  下  $(b : ** : c) = \text{zipmap (常量 } (L . (:** : c))) (\text{下 } b) : ** : \text{zipmap (常量 } (R . (b : ** :))) (\text{下 } c)$  围绕  $v z @ (L (b' : ** : c) :<- vv :: Z (b : ** : c) v x y) = L (\text{dzipmap (常量 } (L . (:** : c))) v \text{ba} : ** : \text{zipmap (常量 } (R . (b : ** :))) (\text{下 } c)) :<- vV v z$  其中  $b = \text{上 } v (b' :<- vv :: Z b v x y)$   $\text{ba} = \text{围绕 } v (b' :<- vv :: Z b v x y)$  围绕  $v z @ (R (b : ** : c') :<- vv :: Z (b : ** : c) v x y) = R (\text{zipmap (常量 } (L . (:** : c))) (\text{下 } b) : ** : \text{dzipmap (常量 } (R . (b : ** :))) v \text{ca}) :<- vV v z$  其中  $c = \text{上 } v (c' :<- vv :: Z c v x y)$   $\text{ca} = \text{围绕 } v (c' :<- vv :: Z c v x y)$

从概念上讲，它和以前一样，只是增加了更多的官僚主义。我使用预类型孔技术构建了这些，在我尚未准备好处理的地方使用了未定义作为占位符，并在我希望从类型检查器获得有用提示的地方（在任何给定的时间）故意引入了类型错误。你也可以在 Haskell 中体验将类型检查作为视频游戏体验。

递归容器的子节点拉链

$b$  关于  $X$  的偏导数告诉我们如何在一个节点内部向下一步找到一个子节点，因此我们得到了拉链（zipper）的常规概念。

数据  $\text{MuZpr } b \ y = \text{MuZpr } \{ \text{aboveMu} :: [D \ b \ X \ (\text{Mu } b \ y) \ y], \text{hereMu} :: \text{Mu } b \ y \}$

我们可以通过反复插入  $X$  位置，逐步放大直到根部。

$\text{muUp} :: \text{Diff2 } b \Rightarrow \text{MuZpr } b \ y \rightarrow \text{Mu } b \ y$   
 $\text{muUp } (\text{MuZpr } \{ \text{aboveMu} = [], \text{hereMu} = t \}) = t$   
 $\text{muUp } (\text{MuZpr } \{ \text{aboveMu} = (dX : dXs), \text{hereMu} = t \}) = \text{muUp } (\text{MuZpr } \{ \text{aboveMu} = dXs, \text{hereMu} = \text{In } (\text{up } VX \ (dX :<- XX \ t)) \})$

但我们需要 *element*-拉链。

双函子不动点的元素拉链

每个元素都位于某个节点内部。该节点位于一叠  $X$  导数之下。但是该节点中元素的位置由  $Y$  导数给出。我们得到

数据  $\text{MuCx } b \ y = \text{MuCx } \{ \text{aboveY} :: [D \ b \ X \ (\text{Mu } b \ y) \ y], \text{belowY} :: D \ b \ Y \ (\text{Mu } b \ y) \ y \}$

$\text{instance Diff2 } b \Rightarrow \text{Functor } (\text{MuCx } b) \text{ where fmap } f \ (\text{MuCx } \{ \text{aboveY} = dXs, \text{belowY} = dY \}) = \text{MuCx } \{ \text{aboveY} = \text{map } (\text{bimap } (\text{fmap } f) \ f) \ dXs, \text{belowY} = \text{bimap } (\text{fmap } f) \ f \ dY \}$

我大胆地断言

实例  $\text{Diff2 } b \Rightarrow \text{Diff1 } (\text{Mu } b)$  其中 类型  $\text{DF } (\text{Mu } b) = \text{MuCx } b$

但在我开发操作之前，我需要一些零碎的东西。我可以在函子拉链和二函子拉链之间交换数据，如下所示：

`zAboveY :: ZF (Mu b) y -> [D b X (Mu b y) y] -- 我之上的 ‘X’ 导数的栈` `zAboveY (d :-: y) = aboveY d`

`zZipY :: ZF (Mu b) y -> Z b Y (Mu b y) y -- 这个 ‘Y’ 拉链，其中我在` `zZipY (d :-: y) = belowY d :-: Y Y y`

这就足以让我定义：

`upF z = muUp (MuZpr { aboveMu = zAboveY z, hereMu = In (up VY (zZipY z)) })`

也就是说，我们先在元素所在的节点处向上重组，把一个元素拉链转换为子节点拉链，然后如上所述一路退回到最外层。

接下来，我说

下F = y在下 []

从空栈开始向下，并定义一个辅助函数，该函数从任意栈的栈底开始反复向下：

`yOnDown :: Diff2 b => [D b X (Mu b y) y] -> Mu b y -> Mu b (ZF (Mu b) y) yOnDown dXs (在 b) = 在 (语境化 dXs (下 b))`

现在，`down b` 只会把我们带到节点内部。我们需要的拉链还必须携带该节点的上下文。这正是 `contextualise` 所做的事情：

`情境化 :: (Bifunctor c, Diff2 b) => [D b X (Mu b y) y] -> c (Z b X (Mu b y) y) (Z b Y (Mu b y) y) -> c (Mu b (ZF (Mu b) y)) (ZF (Mu b) y)` `情境化 dXs = bimap (\ (dX :-: XX t) -> yOnDown (dX : dXs) t) (\ (dY :-: YY y) -> MuCx { aboveY = dXs, belowY = dY } :-: y)`

对于每个Y位置，我们必须提供一个元素拉链，因此我们很高兴知道整个上下文dXs回溯到根节点，以及描述元素在其节点中位置的dY。对于每个X位置，还有一个进一步的子树需要探索，因此我们扩展栈并继续前进！

这就只剩下转移焦点这件事了。我们可以停留在原地，或者从当前的位置往下走，或者往上走，或者先往上再沿着另一条路径往下走。开始吧。

`aroundF z@(MuCx { aboveY = dXs, belowY = dY } :-: _) = MuCx { aboveY = yOnUp dXs (在 (向上 VY (zZipY z))), belowY = contextualize dXs (cxZ $ 围绕 VY (zZip Y z)) } :-: z`

一如既往，现有元素会被其整个拉链所替换。对于 `belowY` 部分，我们查看在现有节点中还能到哪里去：我们会发现要么是可选的元素 Y 位置，要么是需要进一步探索的 X 子节点，因此我们将它们置于上下文中。对于 `aboveY` 部分，在重新组装我们正在访问的节点之后，必须沿着 X 导数的栈向上回溯。

`yOnUp :: Diff2 b => [D b X (Mu b y) y] -> Mu b y -> [D b X (Mu b (ZF (Mu b) y)) (ZF (Mu b) y)] yOnUp [] t = [] yOnUp (dX : dXs) (t :: Mu b y) = 情境化 dXs (cxZ $ 围绕 VX (dX :-: XX t)) : yOnUp dXs (In (up VX (dX :-: XX t)))`

在前进的每一步，我们要么转向附近的别处，要么继续向上。就这样！我没有给出这些法则的形式化证明，但在我看来，这些操作在遍历该结构时似乎谨慎而正确地保持了上下文。

我们学到了什么？

可微性引入了事物在其上下文中的概念，进而引入了一个共单模结构，其中 `extract` 给出事物，而 `duplicate` 探索上下文以寻找其他事物进行上下文化。如果我们为节点拥有适当的微分结构，我们可以为整个树形结构发展微分结构。

哦，将每个类型构造器的不同元数单独处理是显而易见的糟糕做法。更好的方法是处理索引集之间的函子。

```
f :: (i -> *) -> (o -> *)
```

在这里，我们构造了不同种类的结构来存储不同种类的元素。这些是在雅可比构造下的 *closed*

```
J f :: (i -> *) -> ((o, i) -> *)
```

每个结果的  $(o, i)$ -结构都是一个偏导数，告诉你如何在  $o$ -结构中制作一个  $i$ -元素孔。但那是依赖类型的乐趣，留到下次再说。

## 6.8 可遍历性与拉链：必要性与充分性

每一个 `Traversable` 函子都是一个具有有限个元素位置的容器。为了组合在每个元素处进行计算所产生的效果，这些位置必须是有限的。因此，例如，无限的 `Stream` 函子不是 `Traversable` 的，因为无法提供一个可靠的函数来将 `Maybe` 贯穿其中。我们将需要

```
序列 :: Stream (Maybe x) -> Maybe (Stream x)
```

但是，如果你想检查流中的每个操作是否成功，你将需要等待很长时间。

拉链对应于识别特定元素位置的能力（这进一步引发了与导数的联系，但那是另一个话题）。要能够将元素重新插入其位置，你需要一种有效的方式来原因位置的相等性。如果只有有限多个位置，那么这一定成立（在没有信息隐藏的情况下）。因此，具有可遍历性无疑足以拥有一个拉链。

然而，这并不是必需的。`Stream` 有一个完全合理的 `Zipper`

```
类型 StreamContext x = ([x], Stream x) 类型 StreamZipper x = (StreamContext x, x)
```

它表示上下文作为一个有限的（好吧，好吧，加几个感叹号）列表，位于选定元素之前，后面是一个无限的流。

无限流中的位置是自然数。自然数具有可判定的相等性，但它们有无穷多个。

tl;dr 有限集意味着可数集，但反之不成立。

## 6.9 如何以惯用法编写这个（有趣的滤波器）函数？

您可以从标准的或应当是标准的部分组装这个函数。接受的答案提供了关于拉链的正确线索。我的答案关于微分和共单子的内容给出了相关操作的一般处理，但让我在这里具体说明。

我将“具有一个元素空洞的列表”的类型定义如下 ws:

数据  $\text{Bwd } x = \text{B0} \mid \text{Bwd } x :< x$  派生 Show 类型  $\text{HoleyList } x$   
 $= (\text{Bwd } x, [x])$

严格来说，我并不需要引入反向列表来做到这一点，但如果必须在脑海中把东西倒过来，我就很容易感到困惑。（巧合的是， $\text{HoleyList}$  是  $[]$  的形式化导出。）

我现在可以定义在其上下文中作为列表元素意味着什么。

类型  $\text{InContext } x = (\text{HoleyList } x, x)$

这个想法是，配对中的第二个组件应该位于反向列表和正向列表之间。我可以定义一个将列表重新连接起来的函数（在通用处理方式中称为  $\text{upF}$ ）。

插头  $:: \text{InContext } x \rightarrow [x]$  插头  $((\text{B0}, \text{xs}), y) = y : \text{xs}$  插头  $((\text{xz} :< x, \text{xs}), y) = \text{插头 } ((\text{xz}, y : \text{xs}), x)$

我也可以定义一个函数，它给出将一个列表拆分的所有方式（一般称为  $\text{downF}$ ）。

选择  $:: [x] \rightarrow [\text{InContext } x]$  选择  $= \text{go B0}$  其中  $\text{go } \text{xz } [] = []$   $\text{go } \text{xz } (x : \text{xs}) = ((\text{xz}, \text{xs}), x) : \text{go } (\text{xz} :< x) \text{xs}$

请注意

$\text{map snd (selections xs)} = \text{xs}$   $\text{map plug (selections xs)} = \text{map (const xs) xs}$

现在我们可以按照Bartek的食谱进行操作了。

$\text{selectModify} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow [a] \rightarrow [[a]]$   $\text{selectModify } p \ f = \text{map (plug . (id *** f)) . filter (p . snd) . selections}$

也就是说：先用测试过滤选择，对当前焦点中的元素应用该函数，然后再把它们拼回去。如果你手头有  $\text{zipper}$  工具，这就是一行代码，而且它应该适用于任何可微函子，而不只是列表！大功告成！

```
> 选择修改 ((1 ==) . (‘取模 ‘2)) (2*) [1..10] [[2,2,3,4,5,6,7,8,9,10]
,[1,2,6,4,5,6,7,8,9,10],[1,2,3,4,10,6,7,8,9,10],[1,2,3,4,5,6,14,8,9,10],[1
,2,3,4,5,6,7,8,18,10]]
```

## 6.10 计算一个依赖于所有先前项的列表项

法定计算警告。这个问题的基本答案涉及到对标准递归方案的专门化。但我有些过于投入，继续深入探索。随着我试图将相同的方法应用于列表以外的结构，事情变得更加抽象。在这个过程中，我最终借鉴了艾萨克·牛顿和拉尔夫·福克斯，并因此设计出了 *alopegmorphism*，这可能是一些新的东西。

但无论如何，类似的东西应该是存在的。它看起来像是 *anamorphism* 或“展开”的特例。我们从库中所谓的  $\text{unfoldr}$  开始。

`unfoldr :: (种子 -> 也许 (值, 种子)) -> 种子 -> [值]`

它展示了如何从一个种子开始生长一个值的列表，反复使用一个称为 *coalge-bra* 的函数。在每一步，余代数决定是停止并返回 `[]`，还是继续通过将一个值添加到从新种子生长的列表中。

`unfoldr coalg s = 情况为 coalg s 时, Nothing -> [] Just (v, s') -> v : unfoldr coalg s'`

在这里，*seed* 类型可以是喜欢的任何东西——任何适合展开过程的本地状态。一个完全合理的种子概念就是“到目前为止的列表”，也许按逆序排列，这样最近添加的元素就离得最近。

`growList :: ([value] -> 可能值 value) -> [value]` `growList g = unfoldr coalg B0` 其中 `coalg vz = case g vz of -- 我说“vz”，不是“vs”，以记住它是反向的 Nothing -> Nothing Just v -> Just (v, v : vz)`

在每一步，我们的 *g* 操作会查看我们已经拥有的值的上下文，并决定是否添加另一个：如果是，新的值将成为列表的头部，并成为新上下文中的最新值。

因此，这个 *growList* 在每一步将你之前的结果列表交给你，准备好进行 *zipWith (\*)*。反转操作对于卷积来说相当方便，因此我们可能在看类似这样的东西：

`ps = growList $ \ pz -> Just (sum (zipWith (*) sigmas pz) `div` (length pz + 1)) sigmas = [sigma j | j <- [1..]]`

也许？

递归方案？对于列表，我们有一个特殊的逆变形案例，其中种子是我们迄今为止构建的上下文，一旦我们说明如何再构建一点，我们就知道如何通过相同的方式扩展上下文。看出它如何在列表中工作并不难。但对于一般的逆变形，它是如何工作的呢？这就是事情变得复杂的地方。

我们构建可能是无限的值，其节点形状由某个函子 *f* 给定（当我们“打结”时，它的参数会变成“子结构”）。

新类型 `Nu f = 在 (f (Nu f))`

在一个反变换中，余代数使用种子来选择最外层节点的形状，并为子结构填充种子。(协)归纳地，我们将反变换映射过去，将这些种子生长成子结构。

`ana :: 函子 f => (seed -> f seed) -> seed -> Nu f` `ana coalg s = In (fmap (ana coalg) (coalg s))`

让我们从 *ana* 重建 *unfoldr*。我们可以从 *Nu* 和一些简单的部分构建许多普通的递归结构：*polynomial Functor kit*。

新类型 `K1 a x = K1 a -- 常量 (标签)`      新类型 `x = I x -- 子结构位置数据 (f :+ g) x = L1 (f x) | R1 (g x) -- 选择 (像 Either)`      数据 `f (*: g) x = f x *: g x -- 配对 (像 (,))`

带有函数对象实例

实例 Functor (K1 a) where fmap f (K1 a) = K1 a    实例 Functor I where fmap f (I s) =  
 I (f s)    实例 Functor f, Functor g => Functor (f :+: g) where fmap h (L1 fs) = L1 (fm  
 ap h fs) fmap h (R1 gs) = R1 (fmap h gs)    实例 Functor f, Functor g => Functor (f :  
 : g) where fmap h (fx :\*: gx) = fmap h fx :\*: fmap h gx

对于值的列表，节点形状函子是

类型 ListF 值 = K1 () :+: (K1 值 :\*: I)

意思是“要么是一个无趣的标签（表示 nil），要么是一个值标签和一个子列表组成的（cons）对”。ListF 值余代数的类型变为

种子 -> (K1 () :+: (K1 值 :\*: I)) 种子

这是同构的（通过在种子上“评估”多项式 ListF 值）

种子 -> Either () (值, 种子)

离...只有一线之差

种子 -> 也许 (值, 种子)

展开器期望的。你可以像这样恢复一个普通的列表

列表 :: Nu (ListF a) -> [a] 列表 (In (L1 \_)) = [] 列表 (In (R1 (K1 a  
 :\*: I as))) = a : 列表 as

现在，我们如何生成一些通用的 Nu f 呢？一个好的开始是选择最外层节点的 *shape*。类型为 f ( ) 的值仅给出了一个节点的形状，子结构位置上有微不足道的存根。实际上，为了生成我们的树，我们基本上需要某种方法来选择“下一个”节点形状，基于我们目前所处的位置和已完成的工作。我们应该预期

增长 :: (..我所在的一个正在构建中的 Nu f.. -> f ()) -> Nu f

注意，对于增长的列表，我们的 step 函数返回一个 ListF 值 ()，它与 Maybe 值是同构的。

但是我们如何表达我们在 Nu f 中的位置呢？我们将从结构的根节点开始，经过若干节点，因此我们应该预期有一堆层次。每一层应该告诉我们 (1) 它的形状，(2) 我们当前所在的位置，以及 (3) 我们当前位置左侧已经构建的结构，但我们预期在我们尚未到达的位置仍然会有未完成的部分。换句话说，这是我在2008年POPL论文中关于小丑和小丑角色的 *dissection* 结构的一个例子。

剖分算子将一个函子 f（视为元素的容器）转化为一个双函子 Diss f，其中包含两种不同类型的元素：位于 f 结构中“光标位置”左侧的 (clowns) 和右侧的 (jokers)。首先，我们来看一下 Bifunctor 类型类及其一些实例。

类 Bifunctor b 其中

bimap :: (c -> c') -> (j -> j') -> b c j -> b c' j'

新类型 K2 a c j = K2 a 数据 (f :+: g) c j = L2 (f c j) | R2 (g c j)

数据  $(f :: g) \ c \ j = f \ c \ j :: g \ c \ j$  新类型 小丑  $f \ c \ j = \text{小丑} (f \ c)$  新类型 小丑  $f \ c \ j = \text{小丑} (f \ j)$

实例 Bifunctor (K2 a) 在哪里

$\text{bimap } h \ k \ (K2 \ a) = K2 \ a$  实例  $(\text{Bifunctor } f, \text{Bifunctor } g) \Rightarrow \text{Bifunctor } (f :++: g)$  其中  $\text{bimap } h \ k \ (L2 \ fcj) = L2 \ (\text{bimap } h \ k \ fcj) \ \text{bimap } h \ k \ (R2 \ gcj) = R2 \ (\text{bimap } h \ k \ gcj)$  实例  $(\text{Bifunctor } f, \text{Bifunctor } g) \Rightarrow \text{Bifunctor } (f :: g)$  其中  $\text{bimap } h \ k \ (fcj :: gcj) = \text{bimap } h \ k \ fcj :: \text{bimap } h \ k \ gcj$  实例  $\text{Functor } f \Rightarrow \text{Bifunctor } (\text{Clowns } f)$  其中  $\text{bimap } h \ k \ (\text{Clowns } fc) = \text{Clowns } (\text{fmap } h \ fc)$  实例  $\text{Functor } f \Rightarrow \text{Bifunctor } (\text{Jokers } f)$  其中  $\text{bimap } h \ k \ (\text{Jokers } fj) = \text{Jokers } (\text{fmap } k \ fj)$

注意,  $\text{Clowns } f$  是一个双函子, 它等价于一个只包含  $\text{clowns}$  的  $f$  结构; 而  $\text{Jokers } f$  只包含  $\text{jokers}$ 。如果你对为了得到双函子的那一整套函子配套而反复重复所有  $\text{Functor}$  的配套细节感到厌烦, 那么你有理由厌烦: 如果我们抽象掉元数, 转而处理 *indexed* 集合之间的函子, 这个过程就会省力得多, 不过那又是另外一个故事了。

我们定义 *dissection* 为一个类, 它将双函子与函子关联。

类  $(\text{Functor } f, \text{Bifunctor } (\text{Diss } f)) \Rightarrow \text{Dissectable } f$  其中 类型  $\text{Diss } f :: * \rightarrow * \rightarrow * \rightarrow * \rightarrow *$   $\text{rightward} :: \text{Either } (f \ j) \ (\text{Diss } f \ c \ j, c) \rightarrow \text{Either } (j, \text{Diss } f \ c \ j) \ (f \ c)$

类型  $\text{Diss } f \ c \ j$  表示一个  $f$ -结构, 其中在一个元素位置上有一个“空洞”或“游标位置”, 并且在空洞左侧的位置上, 我们有  $c$  中的“小丑”, 而在右侧我们有  $j$  中的“傻瓜”。(这个术语来源于 *Stealer's Wheel* 歌曲 “Stuck in the Middle with You”。)

该类中的关键操作是向右的同构, 它告诉我们如何从任一位置开始向右移动一个位置

- 在一个充满小丑的整体结构的左侧, 或者
- 在结构中有一个洞, 再加上一个小丑放入洞中

并到达任一

- 结构中的一个孔, 与从中出来的小丑一起, 或者
- 整个充满小丑的结构右侧。

艾萨克·牛顿偏爱剖分, 但他将它们称为 *divided differences*, 并将其定义在实值函数上, 以获得曲线上两点之间的斜率, 因此

$$\text{divDiff } f \ c \ j = (f \ c - f \ j) / (c - j)$$

他利用它们对任何旧函数等进行最佳多项式逼近。展开并相乘

$$\text{divDiff } f \ c \ j * c - j * \text{divDiff } f \ c \ j = f \ c - f \ j$$

然后通过两边都加上相同的值来消除减法。

$$f \ j + \text{divDiff } f \ c \ j * c = f \ c + j * \text{divDiff } f \ c \ j$$

和你得到了右向同构。

我们可能通过观察这些实例来培养对这些事物的直觉，然后我们可以回到我们原来的问题。

一个无聊的旧常数，其分割差为零。

实例 可解剖 (K1 a) 其中 类型  $\text{Diss (K1 a) = K2 空 rightward}$   
 $(\text{左 (K1 a)}) = (\text{右 (K1 a)}) \text{ rightward } (\text{右 (K2 v, \_)} ) = \text{荒谬 v}$

如果我们从左开始并向右走，我们跳过整个结构，因为没有元素位置。如果我们从一个元素位置开始，那么有人在撒谎！

恒等函子只有 *one* 个位置。

实例 可分解 I 其中

类型  $\text{Diss I = K2 () 向右 } (\text{左 (I j)}) = \text{左 (j, K2 ()) 向右 } (\text{右 (K2 (), c)}) = \text{右 (I c)}$

如果我们从左边开始，我们到达位置并且小丑跳了出来；推入一个小丑，我们最终到达右边。

对于求和，结构是继承的：我们只需要正确地进行去标签和重新标签。

实例 (可分解 f, 可分解 g) => 可分解 (f :+: g) 其中类型  $\text{Diss (f :+: g) = Diss f :+: Diss g}$

右侧 x = 情况 x 的

$\text{左 (L1 fj) -> ll (向右 (左 fj)) 右 (L2 df, c) -> ll (向右 (右 (df, c))) 左 (R1 gj)}$   
 $\text{-> rr (向右 (左 gj)) 右 (R2 dg, c) -> rr (向右 (右 (dg, c)))}$

在哪里

$\text{ll (左 (j, df)) = 左 (j, L2 df) ll (右 fc) =}$   
 $\text{右 (L1 fc) rr (左 (j, dg)) = 左 (j, R2 dg) r}$   
 $\text{r (右 gc) = 右 (R1 gc)}$

对于产品，我们必须位于一对结构中的某个位置：要么我们位于小丑和愚人之间的左侧，并且右侧结构全是愚人，要么左侧结构全是小丑，而我们位于小丑和愚人之间的右侧。

实例 (可解剖 f, 可解剖 g) => 可解剖 (f :\*: g) 其中类型  $\text{Diss (f :*: g) = (Diss f :*: 玩笑 g) :+: (小丑 f :*: Diss g)}$  向右 x = 情况 x 为 左 (fj :\*: gj) -> ll (向右 (左 fj)) gj 右 (L2 (df :\*: 玩笑 gj), c) -> ll (向右 (右 (df, c))) gj 右 (R2 (小丑 fc :\*: dg), c) -> rr fc (向右 (右 (dg, c))) 其中

$\text{ll (左 (j, df)) gj = 左 (j, L2 (df :*: 小丑 gj)) ll (右 fc) gj = rr fc (向右 (左 gj)) -- (!) rr f}$   
 $\text{c (左 (j, dg)) = 左 (j, R2 (小丑 fc :*: dg)) rr fc (右 gc) = 右 (fc :*: gc)}$

右向逻辑确保我们先通过左侧结构，然后完成后，再开始处理右侧结构。标记为 (!) 的那一行是关键时刻，在这个位置，我们从左侧结构的右侧出来，然后进入右侧结构的左侧。

胡埃特关于数据结构中“左”与“右”光标移动的概念源自可解性（如果你完成了右向同构与其左向对应物的结合）。f 的 *derivative* 只是当小丑与小丑牌之间的差异趋近于零时的极限，或者对我们来说，当你在光标两侧拥有相同类型的元素时所得到的结果。

此外，如果将 clowns 取为零，你会得到

向右 :: Either (f x) (Diss f Void x, Void) -> Either (x, Diss f Void x) (f Void)

但我们可以去除不可能的输入情况，以得到

类型 Quotient f x = Diss f Void x 最左 :: f x -> Either (x, Quotient f x) (f Void)  
d) 最左 = 向右 . 左

这告诉我们，每个 f 结构要么有一个最左侧元素，要么完全没有，这个结果我们在学校里学到的就是“余数定理”。商算符的多变量版本是“导数”，这是 Brzozowski 应用于正则表达式的。

但是 *our* 的一个特例是 Fox 的导数（这是我从 Dan Piponi 那里了解到的）：

类型 Fox f x = Diss f x ()

这就是带有位于光标右侧的存根的 f-结构类型。Now 我们可以给出我们通用 grow 算子的类型。

生长 :: 可分解的 f => ([Fox f (Nu f)] -> f ()) -> Nu f

我们的“上下文”是一个层堆，每一层的左侧有完全增长的数据，右侧有存根。我们可以直接实现 grow，如下所示：

生长 g = go [] 其中 go stk = 在 (walk (rightward (Left (g stk)))) 其中 walk (Left (), df) = walk (rightward (Right (df, go (df : stk)))) walk (Right fm) = fm

当我们到达每一个位置时，我们取出的 joker 只是一个存根，但它的上下文告诉我们该如何扩展栈，从而生长出树的一个子结构，这就给了我们向右移动所需要的 clown。一旦我们用树填充了所有的存根，就完成了！

但是这里有个转折：grow 作为一个 anamorphism 并不像看起来那么简单。给每个节点的最左边子节点提供“种子”很容易，因为我们右侧只有存根。然而，要给右侧下一个子节点提供种子，我们不仅需要最左边的种子——我们还需要从它生长出来的树！anamorphism 模式要求我们在生长任何子结构之前，必须先给出所有子结构的种子。我们的 growList 之所以是一个 anamorphism，仅仅是因为列表节点有 *at most one* 子节点。

所以说，归根结底这是一种新的东西，从无到有地生长，但又允许在某一层上的后续生长依赖于更早的树，而 Fox 导数捕捉了“我们尚未着手之处的残桩”这一想法。也许我们应该把它称为一个 *alopegmorphism*，源自希腊语中表示“狐狸”的  $\alpha\lambda\omega\pi\eta\xi$ 。

## 6.11 合理的 Comonad 实现（用于非空列表）

非空列表通过两种标准构造产生为两个不同的余单子。首先，*cofree comonad* 给出如下。

数据  $\text{Cofree } f \, x = x : \& f (\text{Cofree } f \, x)$  -- 每个节点都标记有一个  $x$

实例 传递器  $f \Rightarrow$  传递器  $(\text{Cofree } f)$  其中  $\text{fmap } f \, (x : \& \text{fcx}) = f \, x : \& \text{fmap } f \, \text{fcx}$

instance Functor  $f \Rightarrow$  Comonad  $(\text{Cofree } f)$  where  $\text{extract } (x : \& \_) = x$  -- 获取顶层节点的标签  $\text{duplicate } \text{cx} @ (\_ : \& \text{fcx}) = \text{cx} : \& \text{fmap duplicate } \text{fcx}$

非空列表可以表示为

类型  $\text{Nellist1} = \text{Cofree Maybe}$

因此，它们是自动的共单代数。这给你“尾巴”共单代数。

与此同时，将一种结构分解为“元素拉链”会诱导出余单子结构。正如我曾经长篇详述的那样，

可微性就是在拉链上进行这一系列操作（将单独的元素从它们的上下文中提取出来并“聚焦”）

类(函子  $f$ , 函子  $(\text{DF } f) \Rightarrow \text{Diff1 } f$  其中

类型  $\text{DF } f :: * \rightarrow * \text{ upF} :: \text{ZF } f \, x \rightarrow f \, x$  -- 去焦点  $\text{downF} :: f \, x \rightarrow \text{ZF } f \, x$  -- 查找所有聚焦方式  $\text{aro}$

$\text{undF} :: \text{ZF } f \, x \rightarrow \text{ZF } f (\text{ZF } f \, x)$  -- 查找所有\*重新\*聚焦方式

data  $\text{ZF } f \, x = (\_ : \& \_)$  {  $\text{cxF} :: \text{DF } f \, x$ ,  $\text{elF} :: x$  }

所以我们得到一个函子和一个共单子

实例  $\text{Diff1 } f \Rightarrow$  函子  $(\text{ZF } f)$  其中  $\text{fmap } f \, (\text{df} : \& \_ : x) = \text{fmap } f \, \text{df} : \& \_ : f \, x$

实例  $\text{Diff1 } f \Rightarrow$  共同单子  $(\text{ZF } f)$  其中提取 =  $\text{elF}$  重复 =  $\text{aro}$

原则上，非空列表也可以通过这种构造产生。问题在于，被求导的函子在 Haskell 中并不容易表示，尽管导数是合理的。让我们疯狂一点……

非空列表相当于 ZF 事物  $x$ ，其中  $\text{DF 事物} = []$ 。我们能整合列表吗？在代数上玩弄可能会给我们一个线索。

$[x] = \text{Either } () \, (x, [x]) = 1 + x * [x]$

所以作为幂级数，我们得到

$[x] = \text{求和}(n :: \text{Nat}). x^n$

我们可以将幂级数积分

积分  $[x] \, dx = \text{求和}(n :: \text{自然数}). x^{(n+1)} / (n+1)$

这意味着我们得到某种大小为  $(n+1)$  的任意元组，但我们必须根据某种关系对它们进行标识，其中等价类的大小为  $(n+1)$ 。一种方法是根据旋转对元组进行标识，这样你就不知道  $(n+1)$  个位置中的哪个是“第一个”。

也就是说，列表是非空循环的导数。想象一群人在圆桌旁玩牌（可能是纸牌接龙）。旋转桌子，你会得到同样的一群人。

扑克牌。但一旦你指定了 *dealer*，你就固定了其他玩家的顺序，从发牌员左侧按顺时针方向排列。

两个 *s* 标准构造；两个相同函数的共单模 或。

（在我之前的评论中，我提到过存在多个单子的可能性。这有点复杂，但这里给出一个起点。每个单子 *m* 也都是 *applicative*，而 *applicative* 定律使得 *m ()* 构成一个幺半群。相应地，*m ()* 的每一种幺半群结构至少都会给 *m* 提供一个单子结构的候选。在 *writer* 单子 *(,) s* 的情况下，我们得到的结论是：单子的候选正是 *(s,())* 上的幺半群，而这与 *s* 上的幺半群完全等价。）

编辑非空列表在至少两种不同的方式中也是 *monadic*。

我定义了函子的恒等和配对，如下所示。

新类型 `I x = I x 数据 (f :: g) x = (:&:) {lll :: f x, rrr :: g x}`

现在，我可以如下引入非空列表，然后定义连接。

新类型 `Ne x = Ne ((I :: []) x)`

`cat :: Ne x -> Ne x -> Ne x cat (Ne (I x :: xs)) (Ne (I y :: ys)) = Ne (I x :: (xs ++ y : ys))`

这些是单子的，正如可能为空的列表一样：

`instance Monad Ne where return x = Ne (I x :: []) Ne (I x :: xs) >>= k = f  
oldl cat (k x) (map k xs)`

然而，*I* 是一个单子：

实例 `Monad I` 其中 `return = I I a`  
`>>= k = k a`

此外，单子在配对下是封闭的：

实例 `(Monad f, Monad g) => Monad (f :: g)` 其中 `return x = return x ::&: return x (fa :: ga)`  
`>>= k = (fa >>= (lll . k)) ::&: (ga >>= (rrr . k))`

所以我们本可以直接写成

新类型 `Ne x = Ne ((I :: []) x)` 衍生 `(Monad, Applicative, Functor)`

但该单子 (*monad*) 的返回给了我们双重视角。

返回 `x = Ne (I x :: [x])`

所以结论是：非空列表有两种余单子方式、两种单子方式、六种应用函子方式，……

（关于这个还有很多要说的，但我必须停在某个地方。）

## 6.12 可表示（或奈皮尔）函子

诺亚对动物们说：“去吧，繁衍生息！”，但蛇说：“我们不能繁衍，因为我们是加法器。”于是诺亚从方舟上取来木材，雕刻成形，说：“我为你们做一张木桌。”

可表示函子有时也被称为“纳佩里安”函子（这是Peter Hancock的术语：Hank与约翰·纳皮尔（对数学名声赫赫）来自同一部分爱丁堡）因为当 $F x \sim T \rightarrow x$ 时，并且记住，从组合学角度看， $T \rightarrow x$ 是“x的T次方”，我们可以看到T在某种意义上是 $\text{Log } F$ 。

首先需要注意的是 $F () \sim T \rightarrow () \sim ()$ 。这告诉我们 *there is only one shape*。提供形状选择的函子不能是Naperian的，因为它们没有提供数据位置的统一表示。这意味着 $[]$ 不是Naperian的，因为不同长度的列表具有由不同类型表示的位置。然而，一个无限的Stream的各个位置由自然数给出。

相应地，给定任何两个F结构，它们的形状必然匹配，因此它们具有合理的配对，从而为我们提供了一个Applicative F实例的基础。

事实上，我们有

$$\begin{array}{c} a \rightarrow p \ x \\ \hline (\text{Log } p, a) \rightarrow x \end{array}$$

使得 $p$ 成为右伴随函子，因此 $p$ 保持所有极限，特别是单元和积，从而使其成为一个单子函子，而不仅仅是一个 $\text{lax}$ 单子函子。也就是说，Applicative的另一种表示方式具有同构的运算。

单元 $:: () \sim p ()$  乘法 $:: (p \ x, p \ y) \sim p \ (x, y)$

让我们为这些东西定义一个类型类。我对它的处理方式与Representable类有些不同。

类Applicative  $p \Rightarrow \text{Naperian } p$  其中

```
类型 Log p logTable :: p (Log p) 投影 :: p x -> Log p -
> x 制表 :: (Log p -> x) -> p x 制表 f = fmap f logTable
e -- 定律1: 投影 logTable = id -- 定律2: 投影 px <$> l
ogTable = px
```

我们有一个类型 $\text{Log } f$ ，表示至少一些 $f$ 内部的位置；我们有一个 $\text{logTable}$ ，在每个位置存储该位置的代表，像是一个包含各地名称的“ $f$ 的地图”；我们有一个 $\text{project}$ 函数，用于提取存储在给定位置的数据。

第一定律告诉我们， $\text{logTable}$ 对所有表示的位置都是准确的。第二定律告诉我们，我们已经表示了*all*这些位置。我们可以推断出

```
制表 (项目 px) = {定义} fmap (项目
px) logTable = {LAW2} px
```

和那个

项目 (制表 f) = { 定义 } 项目 (fmap f logTable) =  
 { 投影的自由定理 } f . 项目 logTable = { 定律1 } f  
 . id = { 组合吸收恒等 } f

我们可以想象一个通用的实例来表示 `Applicative`

实例 `Naperian p => Applicative p` 其中 `pure x = fmap (pure x) logTable pf <$> p`  
`x = fmap (project pf <*> project ps) logTable`

这就是说, `p` 从常规的 `K` 和 `S` 组合子中继承了它自己的 `K` 和 `S` 组合子。

当然, 我们有

实例 `Naperian ((->) r)` 其中类型 `Log ((->) r) = r -- log_x (xf) =`  
`r logTable = id project = ($)`

现在, 所有类似极限的构造都保持了 `Naperian` 性。对数映射将极限事物映射到余极限事物: 它是 *calculates* 左伴随。

我们有终端对象和积积。

数据 `K1 x = K1` 实例 `Applicative K1` 其中 `pure x = K1 K1 <*> K`  
`1 = K1` 实例 `Functor K1` 其中 `fmap = (<*>) . pure`

实例 `Naperian K1`, 其中类型 `Log K1 = Void -- "log(1)`  
 是 `0" logTable = K1` 项目 `K1` 无意义 = 荒谬的无意义

数据 `(p * q) x = p x :: q x` 实例 `(Applicative p, Applicative q) => Applicative (p * q)` 其中 `pure x =`  
`pure x :: pure x (pf :: qf) <*> (ps :: qs) = (pf <*> ps) :: (qf <*> qs)` 实例 `(Functor p, Functo`  
`r q) => Functor (p * q)` 其中 `fmap f (px :: qx) = fmap f px :: fmap f qx`

实例 `(Naperian p, Naperian q) => Naperian (p * q)` 其中类型 `Log (p * q) = Either (Log p) (Log q) -- log (p * q)`  
`= log p + log q logTable = fmap Left logTable :: fmap Right logTable project (px :: qx) (Left i) = project px i`  
`project (px :: qx) (Right i) = project qx i`

我们具有恒等性和合成。

数据  $I\ x = I\ x$  实例 `Applicative I` 其中  $\text{pure } x = I\ x$   $I\ f\ <*>\ I\ s = I\ (f\ s)$  实例 `Functor I` 其中  $\text{fmap } f = (<*>) . \text{pure}$

实例 `Naperian I` 其中类型  $\text{Log } I = ()$   $-- \text{log\_x } x = 1$   
 $\text{logTable} = I\ ()$  项目  $(I\ x)\ () = x$

数据  $(p\ <<\ q)\ x = C\ (p\ (q\ x))$  实例  $(\text{Applicative } p, \text{Applicative } q) \Rightarrow \text{Applicative } (p\ <<\ q)$  其中  $\text{pure } x = C\ (\text{pure } (\text{pure } x))$   $C\ pqf\ <*>\ C\ pqs = C\ (\text{pure } (<*>) <*> pqf\ <*> pqs)$  实例  $(\text{Functor } p, \text{Functor } q) \Rightarrow \text{Functor } (p\ <<\ q)$  其中  $\text{fmap } f\ (C\ pqx) = C\ (\text{fmap } (\text{fmap } f)\ pqx)$

实例  $(\text{Naperian } p, \text{Naperian } q) \Rightarrow \text{Naperian } (p\ <<\ q)$  其中类型  $\text{Log } (p\ <<\ q) = (\text{Log } p, \text{Log } q)$   $-- \text{log } (q\ 1\ \text{og } p) = \text{log } p * \text{log } q$   
 $\text{logTable} = C\ (\text{fmap } (\backslash i \rightarrow \text{fmap } (i\ .) \text{logTable}) \text{logTable})$   
 $\text{project } (C\ pqx)\ (i, j) = \text{project } (\text{project } pqx\ i)\ j$

`Naperian` 函子在 *greatest* 固定点下是封闭的，它们的对数是相应的 *least* 固定点。例如，对于流，我们有

```
log_x (Stream x) = log_x (nu y. x * y) =
mu log_xy. log_x (x * y) = mu log_xy. lo
g_x x + log_x y = mu log_xy. 1 + log_xy
= Nat
```

在Haskell中渲染这个有点麻烦，除非引入`Naperian bifunctors`（它有两组位置来处理两种不同的事物），或者（更好）在索引类型上的`Naperian`函子（它们为索引的事物提供索引位置）。不过，容易实现的，且希望能够传达这个概念的是共同自由`comonad`。

`data{-codata-} CoFree p x = x :- p (CoFree p x)` -- 即， $(I * (p\ <<\ \text{CoFree } p))\ x$  实例 `Applicative p`  $\Rightarrow \text{Applicative } (\text{CoFree } p)$  其中  $\text{pure } x = x\ :-\ \text{pure } (\text{pure } x)$   $(f\ :-\ \text{pcf})\ <*>\ (s\ :-\ \text{pcs}) = f\ s\ :-\ (\text{pure } (<*>) <*> \text{pcf } <*> \text{pcs})$  实例 `Functor p`  $\Rightarrow \text{Functor } (\text{CoFree } p)$  其中  $\text{fmap } f\ (x\ :-\ \text{pcx}) = f\ x\ :-\ \text{fmap } (\text{fmap } f)\ \text{pcx}$

实例 `Naperian p`  $\Rightarrow \text{Naperian } (\text{CoFree } p)$  其中类型  $\text{Log } (\text{CoFree } p) = [\text{Log } p]$  -- 意味着仅有限列表  
 $\text{logTable} = []\ :-\ \text{fmap } (\backslash i \rightarrow \text{fmap } (i\ :) \text{logTable}) \text{logTable}$   
 $\text{project } (x\ :-\ \text{pcx})\ [] = x$   
 $\text{project } (x\ :-\ \text{pcx})\ (i\ : \text{is}) = \text{project } (\text{project } \text{pcx } i)\ \text{is}$

我们可以取流 = CoFree I, 从而得到

日志流 = [日志 I] = [()] ~ = 自然数

现在, 函子  $p$  的导数  $Dp$  给出了其单孔上下文的类型, 告诉我们: i)  $p$  的形状; ii) 孔的位置; iii) 不在孔中的数据。如果  $p$  是 Naperian 的, 则形状没有选择, 因此在非孔位置放入平凡数据, 我们发现得到的仅仅是孔的位置。

$Dp () \sim$  对数  $p$

关于这个连接的更多信息可以在我关于字典树的回答中找到。

无论如何, Naperian 确实是一个有趣的苏格兰地方名字, 用于表示可表示对象, 它们是可以构建对数表的事物: 这些构造完全由投影特征决定, 提供了没有“形状”选择的选项。

## 6.13 前缀树作为纳皮尔函子; 通过其导数进行匹配

编辑: 我想起了一个关于对数和导数的非常有用的事实, 我是在一位朋友的沙发上宿醉得令人作呕时发现的。遗憾的是, 那位朋友 (已故而伟大的 Kostas Tourlas) 已经不再与我们同在了, 但我通过在另一位朋友的沙发上宿醉得令人作呕来纪念他。

让我们回顾一下字典树 (tries)。(在2000年代初, 我的很多同事都在研究这些结构: 在这方面, Ralf Hinze、Thorsten Altenkirch 和 Peter Hancock 会立刻浮现在脑海中。) 真正发生的事情是, 我们在计算类型  $t$  的指数, 并记住  $t \rightarrow x$  是书写  $x \uparrow t$  的一种方式。

也就是说, 我们期望为类型  $t$  配备一个函子  $\text{Expo } t$ , 使得  $\text{Expo } t \ x$  表示  $t \rightarrow x$ 。我们还应该期望  $\text{Expo } t$  是应用型的 (迅速的)。编辑: Hancock 称这种函子为“纳皮尔函子”, 因为它们有对数, 并且它们像函数一样是应用型的,  $\text{pure}$  是  $K$  组合子,  $\langle * \rangle$  是  $S$ 。显然,  $\text{Expo } t ()$  必须与  $()$  同构, 其中  $\text{const } (\text{pure } ())$  和  $\text{const } ()$  做了 (没做多少) 工作。

类  $\text{Applicative } (\text{Expo } t) \Rightarrow \text{EXPO } t$  其中

```
type Expo t :: * -> *
appl  :: Expo t x -> (t -> x)      -- trie lookup
abst  :: (t -> x) -> Expo t x      -- trie construction
```

另一种说法是,  $t$  是  $\text{Expo } t$  的 *logarithm*。

(我差点忘了: 微积分爱好者应该检查  $t$  与  $\partial (\text{Expo } t) ()$  同构。这个同构实际上可能相当有用。编辑: 它非常有用, 我们稍后会把它加入 EXPO。)

我们需要一些函子工具包的东西。恒等函子是 `zippiy applicative`。...

```
data I :: (* -> *) where
  I :: x -> I x 派生 (Show, Eq, Functor, Foldable, Traversable)
```

实例  $\text{Applicative } I$  在其中  $\text{pure } x = I \ x$  且  $f$

$\langle * \rangle I \ s = I (f \ s)$

... 其对数是单位类型

```
instance EXPO () where type Expo ()
= I appl (I x) () = x abst f = I (f ())
```

拉链式 Applicative 的积也是拉链式 Applicative。..

数据  $(:*) :: (* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow (* \rightarrow *)$  其中  $(:*) :: f\ x \rightarrow g\ x \rightarrow (f : * : g)\ x$  派生 (Show, Eq, Functor, Foldable, Traversable)

实例  $(\text{Applicative } p, \text{Applicative } q) \Rightarrow \text{Applicative } (p : * : q)$  其中  $\text{pure } x = \text{pure } x : * : \text{pure } x$   $(\text{pf} : * : \text{qf}) < * > (\text{ps} : * : \text{qs}) = (\text{pf} < * > \text{ps}) : * : (\text{qf} < * > \text{qs})$

……并且它们的对数是和。

实例  $(\text{EXPO } s, \text{EXPO } t) \Rightarrow \text{EXPO } (\text{Either } s\ t)$  其中 类型  $\text{Expo } (\text{Either } s\ t) = \text{Expo } s : * : \text{Expo } t$   $\text{appl } (\text{sf} : * : \text{tf}) (\text{Left } s) = \text{appl } \text{sf } s$   $\text{appl } (\text{sf} : * : \text{tf}) (\text{Right } t) = \text{appl } \text{tf } t$   $\text{abst } f = \text{abst } (f . \text{Left}) : * : \text{abst } (f . \text{Right})$

可 zip 的 Applicative 的组合仍然是以 zip 方式的 Applicative。..

数据  $(:<:) :: (* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow (* \rightarrow *)$  其中  $C :: f\ (g\ x) \rightarrow (f : <: g)\ x$  派生 (Show, Eq, Functor, Foldable, Traversable)

实例  $(\text{Applicative } p, \text{Applicative } q) \Rightarrow \text{Applicative } (p : <: q)$  , 其中  $\text{pure } x = C\ (\text{pure } (\text{pure } x))$   $C\ \text{pqf} < * > C\ \text{pqs} = C\ (\text{pure } (< * >) < * > \text{pqf} < * > \text{pqs})$

它们的对数是乘积。

实例  $(\text{EXPO } s, \text{EXPO } t) \Rightarrow \text{EXPO } (s, t)$  其中 类型  $\text{Expo } (s, t) = \text{Expo } s : <: \text{Expo } t$   $\text{appl } (C\ \text{stf}) (s, t) = \text{appl } (\text{appl } \text{stf } s)\ t$   $\text{abst } f = C\ (\text{abst } \$ \backslash s \rightarrow \text{abst } \$ \backslash t \rightarrow f\ (s, t))$

如果我们开启了足够多的项，我们现在可以写为

新类型 树 = 树 (要么 () (树, 树)) 派生 (显示, 等于) 模式 叶子 = 树 (左 () 模式 节点  $l\ r = \text{树 } (\text{右 } (l, r))$ )

新类型  $\text{ExpoTree } x = \text{ExpoTree } (\text{Expo } (\text{Either } ()\ (\text{Tree}, \text{Tree}))\ x)$  派生 (Show, Eq, Functor, Applicative)

实例 EXPO 树, 其中类型  $\text{Expo Tree} = \text{ExpoTree } \text{appl } (\text{ExpoTree } f)\ (\text{Tree } t) = \text{appl } f\ t$   $\text{abst } f = \text{ExpoTree } (\text{abst } (f . \text{Tree}))$

树图是一种在问题中提到的类型，作为

```
data TreeMap a = TreeMap { tm_leaf :: Maybe a, tm_no
de :: TreeMap (TreeMap a) }
```

正是Expo Tree（可能是a），其中lookupTreeMap是翻转应用。

现在，鉴于树和树  $\rightarrow x$  是截然不同的事物，我觉得想要代码在“两个”上都能工作是很奇怪的。树等式测试只是查找的一个特殊情况，区别在于树等式测试是一个作用于树的普通函数。然而，有一个巧合：为了测试相等性，我们必须将每棵树转变为它自己的自我识别器。编辑：这正是 log-diff iso 所做的。

引发等式测试的结构是某种*matching*的概念。像这样：

```
类 Matching a b, 其中 类型 Matched a b :: * matched :: Matc
hed a b -> (a, b) match :: a -> b -> Maybe (Matched a b)
```

也就是说，我们期望“Matched a b”以某种方式表示一个匹配的a和b对。我们应该能够提取出这个对（忘记它们匹配的事实），并且我们应该能够拿任何一对并尝试匹配它们。

不出所料，我们也可以对单位类型这样做，而且相当成功。

```
instance Matching () () where type Matche
d () () = () matched () = ((), ()) match
() () = Just ()
```

对于乘积，我们按分量进行处理，唯一的风险是分量不匹配。

实例 (Matching s' , Matching t t' ) => 匹配 (s, t) (s' , t' ) 其中 类型 Matched (s, t) (s' , t' ) = ( 已匹配 s s' , 已匹配 t t' ) 匹配了 (ss' , tt' ) = ( (s, t), (s' , t' ) ) 其中 (s, s' ) = 匹配了 ss' (t, t' ) = 匹配了 tt' 匹配 (s, t) (s' , t' ) = (,) <\$> 匹配 s s' <\*> 匹配 t t'

总和提供了一些不匹配的可能性。

实例 (Matching s s' , Matching t t' ) => Matching (Either s t) (Either s' t' ) 其中 类型 Matched (Eith er s t) (Either s' t' ) = Either (Matched s s' ) (Matched t t' ) matched (Left ss' ) = (Left s, Left s' ) 其中 (s, s' ) = matched ss' matched (Right tt' ) = (Right t, Right t' ) 其中 (t, t' ) = matched tt' m atch (Left s) (Left s' ) = Left <\$> match s s' match (Right t) (Right t' ) = Right <\$> match t t' match \_ \_ = Nothing

有趣的是，我们现在就能同样轻松地获得用于树的相等性测试

```
实例 匹配 树 树, 其中 类型 Matched 树 树 = 树 已匹配 t = (t, t) 匹配
(树 t1) (树 t2) = 树 <$> 匹配 t1 t2
```

(顺便提一下, 捕捉匹配概念的函子子类, 即 $\{v^*\}$ )

```
class HalfZippable f where -- “half zip” 源自 Roland Backhouse halfZip :: (f a, f b) -> Maybe (f (a, b))
```

却被遗憾地忽视了。从道义上讲, 对于每个这样的  $f$ , 我们应该有

匹配  $(f\ a)\ (f\ b) = f\ (\text{匹配}\ a\ b)$

一个有趣的练习是证明: 如果  $(\text{Traversable}\ f, \text{HalfZippable}\ f)$ , 那么  $f$  上的自由单子具有一个一阶统一算法。) 我想我们可以这样构造“单元素关联列表”:

```
mapOne :: forall a. (树, a) -> Expo 树 (也许 a) mapOne (t, a) = abst f 其中 f :
: 树 -> 也许 a f u = 纯 a <*> 匹配 t u
```

而且我们可以尝试把它们与这个小工具结合起来, 利用所有  $\text{Expo}\ ts$  的敏捷性。...

```
实例 Monoid x => Monoid (ExpoTree x) 其中 mempty = pure mem
pty mappend t u = mappend <$> t <*> u
```

... 但是, 然而,  $\text{Maybe}\ x$  的  $\text{Monoid}$  实例的彻底愚蠢继续使得清晰的设计受到挫折。

我们至少可以应付

```
实例 Alternative m => Alternative (ExpoTree <:> m) 其中 empty = C (纯 empty) C f
<|> C g = C ((<|>) <$> f <*> g)
```

一个有趣的练习是将  $\text{abst}$  与  $\text{match}$  融合在一起, 也许这正是问题真正想要指向的。让我们重构  $\text{Matching}$ 。

```
c class EXPO b => 匹配 a b 其中 类型 Matched a b :: * matched :: Matched a b -> (
a, b) match' :: a -> Proxy b -> Expo b (Maybe (Matched a b))
```

```
data Proxy x = Proxy -- 我还没用上 GHC 8, 而且 Simon 在这里需要搭把手
```

对于  $()$ , 新的内容是

```
instance Matching () () where -- 跳过旧内容 match' () (Proxy ::
Proxy ()) = I (Just ())
```

对于总和, 我们需要标记成功的匹配, 并用极具格拉斯哥特色的纯粹“无”填补不成功的部分。

```
instance (Matching s s', Matching t t') => Matching (Either s t) (Either s' t') where -- 跳过
旧内容 match' (Left s) (Proxy :: Proxy (Either s' t')) = ( (Left <$>) <$> match' s (Proxy ::
Proxy s') ) : pure Nothing match' (Right t) (Proxy :: Proxy (Either s' t')) = pure Nothing :
: ((Right <$>) <$> match' t (Proxy :: Proxy t'))
```

对于成对的情况，我们需要按顺序构建匹配，如果第一个组件失败则提前退出。

我实例 (匹配  $s'$  , 匹配  $t'$  ) => 匹配  $(s, t)$  ( $s'$  ,  $t'$  ) 其中 -- 跳过旧内容  $\text{match}' (s, t) (\text{Poxy} :: \text{Proxy } (s', t')) = C$  (更多  $\langle \$ \rangle \text{match}' s (\text{Poxy} :: \text{Proxy } s')$  ) 其中更多  $\text{Nothing} = \text{pure Nothing}$  更多  $(\text{Just } s) = ( (,) s \langle \$ \rangle ) \langle \$ \rangle \text{match}' t (\text{Poxy} :: \text{Proxy } t')$  )

因此，我们可以看到构造函数与其匹配器的前缀树之间存在关联。

作业：将  $\text{abst}$  与  $\text{match}$  融合，effectively 列出整个过程。

编辑：写匹配时，我们将每个子匹配器停放在对应子结构的前缀树位置。当你想到特别位置的东西时，应该联想到拉链和微积分。让我提醒你。

我们需要函子常数和余积来管理“洞在哪里”的选择。

数据  $K :: * \rightarrow$  ( $*$   $\rightarrow$   $*$ ) where  
 $K :: a \rightarrow K a$  派生 (Show, Eq, Functor, Foldable, Traversable)

数据  $(+:) :: (* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow (* \rightarrow *)$  其中  $\text{Inl} :: f x \rightarrow (f :+: g) x$   $\text{Inr} :: g x \rightarrow (f :+: g) x$  派生 (Show, Eq, Functor, Foldable, Traversable)

现在我们可以定义

c 类 (Functor  $f$ , Functor  $(D f)$ ) => 可微分  $f$  其中类型  $D f :: (* \rightarrow *) \text{plug} :: (D f : * : I) x \rightarrow f x$  -- 现在应该有其他方法，但暂时用  $\text{plug}$  即可

通常的微积分法则适用，组合给 *chain rule* 赋予了空间解释。

实例 可微  $(K a)$  其中 类型  $D (K a) = K$  无 Void 插头 ( $K$

错误  $: * : I x) = K$  (荒谬 错误)

实例 可微  $I$  其中

类型  $D I = K ()$  插头  $(K () : * : I x) = I x$

实例 (可微  $f$ , 可微  $g$ ) => 可微  $(f :+: g)$  其中 类型  $D (f :+: g) = D f :+: D g$

插头  $(\text{Inl } f' : * : I x) = \text{Inl} (\text{插头 } (f' : * : I x))$  插头  $(\text{Inr } g' : * : I x) = \text{Inr} (\text{插头 } (g' : * : I x))$

实例 (可微分  $f$ , 可微分  $g$ ) => 可微分  $(f :<: g)$  其中类型  $D (f :<: g) = (D f :<: g) :+: (f :<: D g) \text{plug} (\text{Inl } (f' : * : g) : * : I x) = \text{plug} (f' : * : I x) :<: g \text{plug} (\text{Inr } (f :<: g') : * : I x) = f :<: \text{plug} (g' : * : I x)$

实例 (Differentiable  $f$ , Differentiable  $g$ ) => Differentiable  $(f :<: g)$  其中

类型  $D (f :<: g) = (D f :<: g) :<: D g$  插头  $((C f' g : * : g') : * : I x) = C (\text{插头 } (f' g : * : I (\text{插头 } (g' : * : I x))))$

e

坚持认为  $\text{Expo } t$  是可微的并不会对我们造成任何损害，因此让我们扩展 EXPO 类。什么是“带孔的 trie”？它是一种 trie，对于所有可能的输入中恰好有一个缺少对应的输出条目。而这正是关键。

类  $(\text{Differentiable } (\text{Expo } t), \text{Applicative } (\text{Expo } t)) \Rightarrow \text{EXPO } t$  其中 类型  $\text{Expo } t :: * \rightarrow *$   $\text{appl} :: \text{Expo } t \rightarrow t \rightarrow t$   $\text{hole} :: t \rightarrow \text{Expo } t$   $\text{hole} :: t \rightarrow \text{D } (\text{Expo } t) ()$   $\text{eloh} :: \text{D } (\text{Expo } t) () \rightarrow t$

现在，hole 和 eloh 将见证该同构。

实例  $\text{EXPO } ()$  其中类型  $\text{Expo } () = I$  -- 跳过旧内容  $\text{hole } () = K ()$   $\text{eloh } (K ()) = ()$

单位情形并不特别令人兴奋，但求和情形开始展现结构：i

实例  $(\text{EXPO } s, \text{EXPO } t) \Rightarrow \text{EXPO } (\text{Either } s \ t)$  其中类型  $\text{Expo } (\text{Either } s \ t) = \text{Expo } s :: \text{Expo } t$   $\text{hole } (\text{Left } s) = \text{Inl } (\text{hole } s :: \text{pure } ())$   $\text{hole } (\text{Right } t) = \text{Inr } (\text{pure } () :: \text{hole } t)$   $\text{eloh } (\text{Inl } (f' :: \_)) = \text{Left } (\text{eloh } f')$   $\text{eloh } (\text{Inr } (\_ :: g')) = \text{Right } (\text{eloh } g')$

看到了吗？左侧映射到一个左侧有孔的字典树；右侧映射到一个右侧有孔的字典树。

现在来看产品。

实例  $(\text{EXPO } s, \text{EXPO } t) \Rightarrow \text{EXPO } (s, t)$  其中 类型  $\text{Expo } (s, t) = \text{Expo } s :: \text{Expo } t$   $\text{hole } (s, t) = C (\text{const } (\text{pure } ()) <\$> \text{hole } s) :: \text{hole } t$   $\text{eloh } (C f' :: g') = (\text{eloh } (\text{const } () <\$> f'), \text{eloh } g')$

用于一对的 trie 是将一个右 trie 嵌入到一个左 trie 中，因此，某个特定对的孔位是通过在对应左元素的特定子 trie 中为右元素创建一个孔来找到的。对于树，我们再加一层封装。

$\text{newtype DExpoTree } x = \text{DExpoTree } (\text{D } (\text{Expo } (\text{Either } () \ (\text{Tree}, \text{Tree}))) \ x)$  deriving (Show, Eq, Functor)

那么，我们如何把一棵树变成它的字典树识别器呢？首先，我们取出它的“除我之外的所有人”字典树，把所有那些输出都填为 False，然后在缺失的条目处填入 True。

$\text{matchMe} :: \text{EXPO } t \Rightarrow t \rightarrow \text{Expo } t \text{ Bool}$   $\text{matchMe } t = \text{plug } ((\text{const False } <\$> \text{hole } t) :: I \text{ True})$

家庭作业提示：D f :: I 是一个 co-monad。  
敬缺席的朋友们！

## 第七章

# 依赖类型的 Haskell

### 7.1 最适合“现实世界”编程的依赖类型语言？

接受的答案包含误导性信息。在Agda中，类型检查是可判定的，除非你关闭了正性/终止性/宇宙检查。此外，无限过程可以在Agda中编程，就像Haskell中的IO过程一样：唯一的限制是，执行时无限过程不能被无限展开 *in the course of typechecking*。你可以在Agda中实现图灵机模拟器：你只不过不能说它一定会终止，或者说服类型检查器以无限的方式运行它。

我同意，依赖类型语言在“真实世界”编程方面仍处于实验阶段。我们目前还不能支持重型开发，但我们可以在那些具有未来眼光的人中维持一个重要的爱好，就像早期的函数式语言一样。

Idris，正如Twey所建议的，是最接近“现实世界”依赖类型语言的候选者。它比Agda更专注于完成任务。我会推荐Agda作为理解依赖类型编程背后思想的更好工具，但Idris则是更实际的选择。

很高兴地说，值得在这个讨论中考虑最近发布的 Haskell 作为候选语言。自 GHC 7.4 起，Haskell 开始支持一种有用的类型级数据概念，并且通过至少使用 *singleton* 技巧（尽管这有些临时解决方案），我们确实可以让类型依赖于运行时值（通过使它们依赖于受约束为等于运行时值的静态变量）。因此，Haskell 是一种真正的“现实世界”语言，目前正处于依赖类型实验的早期阶段。

### 7.2 为什么不使用依赖类型？

依赖类型的 Haskell，现在？

Haskell在某种程度上是一种依赖类型语言。存在类型级数据的概念，得益于DataKinds，现在它们的类型更加合理，并且通过GADT（广义代数数据类型）提供了一种方式，可以为类型级数据提供运行时表示。因此，运行时数据的值实际上会出现在类型中，这就是语言成为依赖类型语言的含义。

简单的数据类型是 *promoted* 到类型级别，因此它们包含的值可以在类型中使用。因此，典型的例子是

```
data Nat = Z | S Nat
```

```
data Vec :: Nat -> * -> * where
```

```
VNil :: Vec Z x VCons :: x -> Vec n x -> Vec (S n) x
```

变得可能，并且随之出现了诸如的定义

```
vApply :: Vec n (s -> t) -> Vec n s -> Vec n t
vApply VNil VNil = VNil
vApply (VCons f fs) ss = VCons (f s) (vApply fs ss)
```

这很好。注意，长度  $n$  在该函数中是一个纯粹的静态量，用来确保输入和输出向量具有相同的长度，尽管这个长度在 `vApply` 的执行过程中并不起任何作用。相比之下，实现一个函数来生成给定  $x$  的  $n$  个副本要棘手得多（即不可能），该函数将是 `vApply` 的 `<*>` 的纯函数。

```
vReplicate :: x -> Vec n x
```

因为在运行时知道需要制作多少份副本是至关重要的。进入单例模式。

```
数据 Natty :: Nat -> * 其中 Zy :: Natty Z Sy
:: Natty n -> Natty (S n)
```

对于任何可提升类型，我们都可以构建一个单例族，它以提升后的类型为索引，其元素是该类型值在运行时的副本。`Natty n` 是类型层级的  $n :: \text{Nat}$  在运行时的副本所对应的类型。我们现在可以写

```
vReplicate :: Natty n -> x -> Vec n x
vReplicate Zy x = VNil
vReplicate (Sy n) x = VCons x (vReplicate n x)
```

所以你有一个与运行时值绑定的类型级值：检查运行时副本可以精炼类型级值的静态知识。尽管术语和类型是分开的，我们仍然可以通过使用单例构造将依赖类型化的方式应用到工作中，类似于一种环氧树脂，创建阶段之间的联系。这距离允许在类型中使用任意运行时表达式还有很长的路要走，但也不算毫无意义。

什么是恶心的？什么是缺失的？

让我们给这项技术施加一些压力，看看什么开始动摇。我们可能会得到这样的想法：单例应该更加隐式地可管理。

```
class Nattily (n :: Nat) where natty :: Natty n
instance Nattily Z where natty = Zy
instance Nattily (S n) where natty = Sy natty
```

允许我们写，例如，

```
实例 Nattily n => 应用型 (Vec n) 其中 pure = vReplicate natty (<*>)
= vApply
```

这可以工作，但现在意味着我们原始的 `Nat` 类型已经衍生出三个副本：一种，单例族和单例类。我们有一个相当笨重的过程来交换显式的 `Natty n` 值和 `Nattily n` 字典。此外，`Natty` 不是 `Nat`：我们在运行时值上有某种依赖关系，但并不是在我们最初想到的类型上。没有任何完全依赖类型语言将依赖类型弄得如此复杂！

与此同时，尽管 `Nat` 可以被提升，但 `Vec` 不能。你不能通过索引类型进行索引。完全依赖类型语言没有这种限制，在我作为一个依赖类型炫耀者的职业生涯中，我学会了在我的讲座中加入二层索引的例子，只是为了教那些将一层索引做得困难但可行的人，不要期望我像纸牌屋一样倒塌。问题出在哪里？等式。GADT 通过将你在给构造函数指定特定返回类型时隐式实现的约束转化为显式的等式要求来工作。像这样。

```
data Vec (n :: Nat) (x :: *) = n Z => VNil | 对于所有 m. n S
    m => VCons x (Vec m x)
```

在我们两个方程中的每一个，两边都有类型 `Nat`。现在尝试对向量索引的内容进行相同的翻译。

```
数据 InVec :: x -> Vec n x -> * 其中 Here :: InVec z (VCons z zs)
After :: InVec z ys -> InVec z (VCons y ys)
```

变成

```
数据 InVec (a :: x) (as :: Vec n x) = 对于 m z (zs :: Vec x m). (n S m, as VCons z zs) => 这里 | 对于 m y z (ys :: Vec x m). (n S m, as VCons y ys) => 之后 (InVec z ys)
```

现在我们在  $\{v^*\}$  和  $\{VCons z zs :: Vec (S m) x\}$  之间形成方程约束，其中两边具有语法上不同（但可证明相等）的种类。GHC 核心目前尚不支持这种概念！

还有什么缺失的吗？嗯，大多数 Haskell 在类型层面上是缺失的。你可以提升的术语语言实际上只有变量和非 GADT 构造函数。一旦你拥有这些，类型家族机制就允许你编写类型层面的程序：其中一些可能类似于你在术语层面上考虑编写的函数（例如，给 `Nat` 配备加法，这样你就可以为 `Vec` 的 `append` 函数提供一个好的类型），但这只是巧合！

另一个在实践中缺失的东西是一个 *library*，它利用了我们通过值对类型进行索引的新能力。在这个勇敢的新世界中，`Functor` 和 `Monad` 会变成什么？我在思考这个问题，但还有很多工作要做。

## 运行类型级程序

Haskell 和大多数依赖类型编程语言一样，具有 *two* 操作语义。运行时系统有一种运行程序的方式（仅限闭合表达式，类型擦除后，经过高度优化），类型检查器也有另一种运行程序的方式（你的类型族，你的“类型类 Prolog”，带有开放表达式）。对于 Haskell，你通常不会混淆这两者，因为被执行的程序是在不同的语言中。依赖类型语言为程序的 *same* 语言提供了独立的运行时和静态执行模型，但不用担心，运行时模型仍然允许你进行类型擦除，实际上也允许你进行证明擦除：这正是 Coq 的 *extraction* 机制所提供的；至少这是 Edwin Brady 的编译器所做的（尽管 Edwin 会擦除不必要的重复值，以及类型和证明）。阶段区分可能不再是 *syntactic category* 的区分，但它依然存在且有效。

依赖类型语言是全局的，允许类型检查器运行程序，免于担心除了长时间等待之外的任何问题。随着 Haskell 变得更加依赖类型，我们

面对静态执行模型应该是什么的问题？一种方法可能是将静态执行限制为总函数，这将允许我们拥有相同的运行自由，但可能迫使我们在数据和共数据之间做出区分（至少对于类型层级的代码），以便我们能够判断是否强制终止或强制生产性。但这不是唯一的方法。我们可以选择一个更弱的执行模型，这个模型不太愿意运行程序，代价是通过计算得出的方程更少。实际上，这正是GHC所做的。GHC核心的类型规则没有提到`running`程序，而只是用于检查方程的证据。当翻译到核心时，GHC的约束求解器尝试运行你的类型层级程序，生成一条小小的银色证据轨迹，证明给定的表达式等于其正常形式。这种证据生成方法有点不可预测，且不可避免地是不完整的：例如，它回避了看起来可怕的递归，这可能是明智的。我们不需要担心的一件事是类型检查器中IO计算的执行：记住，类型检查器不需要赋予`launchMissiles`与运行时系统相同的意义！

### 辛德利-米尔纳文化

Hindley-Milner 类型系统实现了四种彼此不同的区分之间真正令人惊叹的巧合；不幸的文化副作用是，许多人看不清这些区分之间的区别，并假定这种巧合是不可避免的！我在说什么？

- 术语 *vs* 类型
- 显式写出的内容 *vs* 隐式写出的内容
- 运行时存在 *vs* 运行前擦除
- 非依赖抽象 *vs* 依赖量化

我们习惯于编写项，把类型交由推断……然后再被擦除。我们也习惯于对类型变量进行量化，而相应的类型抽象和应用在静态中悄然发生。

你不必偏离纯粹的 Hindley-Milner 太远，这些区分就会失去对齐，而这正是 *no bad thing*。首先，只要我们愿意在少数地方写出它们，就可以拥有更有趣的类型。与此同时，在使用重载函数时我们不必书写类型类字典，但这些字典在运行时肯定是存在的（或被内联）。在依赖类型语言中，我们期望在运行时抹除的不仅仅是类型，但（与类型类一样）有些隐式推断的值不会被抹除。例如，`vReplicate` 的数值参数通常可以从所需向量的类型中推断出来，但在运行时我们仍然需要知道它。

我们应该审查哪些语言设计选择，因为这些巧合不再成立？例如，Haskell 不提供显式实例化 `forall x. t` 量化的方法，这样做是否正确？如果类型检查器不能通过统一 `t` 来推断 `x`，我们就没有其他方式来说明 `x` 必须是什么。

更广泛地说，我们不能将“类型推断”视为一个单一的概念，要么完全具备，要么完全没有。首先，我们需要将“泛化”方面（Milner的“`let`”规则）与“特化”方面（Milner的“`var`”规则）分开，前者在很大程度上依赖于限制可用类型的范围，以确保一个简单的机器能够猜测出某个类型，后者则和你的约束求解器的效果一样有效。我们可以预期，顶层类型将变得更难推断，但内部类型信息将保持相对容易传播。

### Haskell 的下一步

我们看到类型级别和种类级别变得非常相似（它们已经在GHC中共享一个内部表示）。我们完全可以将它们合并。如果可以的话，采用 `* :: *` 会很有趣：我们很久以前就失去了 *logical* 的健全性，当时我们允许了底值，但是 *type* 健全性通常是一个较弱的要求。我们必须检查。如果必须保持类型、种类等级别的区分，至少可以确保类型级别及以上的所有内容总是可以被提升。

如果能直接重用我们在类型层面已经拥有的多态性，而不是在 `kind` 层面重新发明多态性，那就太好了。

我们应当通过允许 *heterogeneous* 方程 `a b`（其中 `a` 和 `b` 的 `kind` 在语法上并不相同，但可以被证明相等）来简化并泛化当前的约束系统。这是一种老技术（出自我上世纪的论文），它使得处理依赖关系容易得多。我们将能够在 GADT 中对表达式施加约束，从而放宽对哪些内容可以被提升的限制。

我们应该通过引入依赖函数类型 `pi x :: s -> t` 来消除单例构造的需求。具有这种类型的函数可以应用于任何类型为 `s` 的表达式，这些表达式存在于类型和术语语言的 *intersection* 中（因此，变量、构造子，后续会有更多内容）。相应的  $\lambda$  表达式和应用在运行时不会被删除，因此我们可以编写

```
vReplicate :: pi n :: Nat -> x -> Vec n xvReplicate Z x = VNilvReplicate (S n) x = VCons x (vReplicate n x)
```

而不需要将 `Nat` 替换为 `Natty`。`pi` 的定义域可以是任何可提升（promotable）的类型，因此如果 GADT 可以被提升，我们就可以编写依赖量词序列（或者如 de Bruijn 所称的“望远镜”）。

```
pi n :: Nat -> pi xs :: Vec n x -> ...
```

到我们需要的任何长度为止。

这些步骤的目的在于通过直接使用更通用的工具来 *eliminate complexity*，而不是将就使用薄弱的工具和笨拙的编码。当前的部分采用使得 Haskell 某种程度上的依赖类型的收益成本高于本应如此。

太难了？

依赖类型让很多人感到不安。它们也让我紧张，但我喜欢这种紧张，或者至少无论如何我很难不紧张。但围绕这个话题存在着相当浓重的无知迷雾，这并没有什么帮助。其中一部分原因在于我们所有人仍然有很多要学。但不那么激进的方法的支持者，众所周知曾在并非总能确保事实完全站在他们一边的情况下，煽动对依赖类型的恐惧。我就不点名了。这些所谓的“类型检查不可判定”“图灵不完备”“没有阶段区分”“没有类型擦除”“到处都是证明”等神话依然存在，尽管它们都是垃圾。

依赖类型程序并不一定必须始终被证明是正确的。人们可以通过提高程序的基本卫生性，强制在类型中加入额外的不变量，而不需要完全指定所有内容。在这个方向上迈出的少许步伐，往往能带来更强的保证，而几乎没有或根本没有额外的证明义务。依赖类型程序并非不可避免地需要 *full* 证明，实际上，我通常把代码中任何证明的存在作为 *question my definitions* 的提示。

因为，正如任何表达能力的提升一样，我们不仅能自由地说出美好的新事物，也能说出糟糕的新事物。例如，定义二叉搜索树的糟糕方式有很多，但这并不意味着不存在好的方式。重要的是不要臆断糟糕的经历无法被改进，即便承认这一点会伤到自尊。依赖型定义的设计是一项需要学习的新技能，而成为一名 Haskell 程序员并不会自动让你成为专家！而且即便有些程序是糟糕的，你又为什么要剥夺他人保持美好的自由呢？

为什么仍然要使用 Haskell？

我非常喜欢依赖类型，但我大多数的黑客项目仍然使用 Haskell。为什么？Haskell 有类型类。Haskell 有有用的库。Haskell 有一个可行的（尽管远非理想的）

对带有效应的编程的处理。Haskell 拥有一个工业级的编译器。依赖类型语言在社区成长和基础设施方面仍处于更早期的阶段，但我们终将到达那一步，并在可能性的层面上实现真正的代际性转变，例如通过元编程和数据类型泛型等方式。不过，你只需看看随着 Haskell 朝向依赖类型迈出的步伐，人们已经在做些什么，就能发现，通过推动现有这一代语言继续向前发展，同样可以获得巨大的收益。

### 7.3 Haskell中简单的依赖类型示例（傻瓜版）。它们在Haskell中实际应用中如何有用？为什么我应该关心依赖类型？

迟到了，这个回答基本上是个厚颜无耻的宣传。

Sam Lindley 和我写了一篇关于 Hasochism 的论文，探讨了在 Haskell 中依赖类型编程的快感与痛苦。文中提供了大量关于 Haskell 中可实现的例子 *now*，并与 Agda/Idris 这一代依赖类型语言进行了对比（包括有利与不利的方面）。

尽管这是篇学术论文，但它是关于实际程序的，你可以从Sam的仓库中获取代码。我们有很多小例子（例如，归并排序输出的有序性），但最后我们用了一个文本编辑器的例子，在这个例子中，我们通过宽度和高度进行索引来管理屏幕几何：我们确保组件是规则的矩形（向量的向量，而不是参差不齐的列表的列表），并且它们能够精确地拼接在一起。

依赖类型的关键力量在于保持独立数据组件之间的一致性（例如，矩阵中的头向量和其尾部的每个向量必须具有相同的长度）。这在编写条件代码时尤为重要。这样的情况（将来将被视为极其天真）是，以下所有内容都是类型保持的重写

- 如果  $b$  那么  $t$  否则  $e \Rightarrow$  如果  $b$  那么  $e$  否则  $t$
- 如果  $b$  则  $t$  否则  $e \Rightarrow t$
- 如果  $b$  那么  $t$  否则  $e \Rightarrow e$

尽管我们假设正在测试  $b$ ，因为它为我们提供了关于接下来应该做什么（甚至是否安全）的有用见解，但这些见解并没有通过类型系统传递：尽管这个观点至关重要，即  $b$  的真值证明了  $t$ ，而其假值证明了  $e$ ，但这一点缺失了。

普通的 Hindley-Milner 确实为我们提供了一种确保一致性的方法。每当我们有一个多态函数时

```
f :: forall a. r[a] -> s[a] -> t[a]
```

我们必须始终如一地实例化：然而，第一个参数固定了  $a$ ，第二个参数必须配合，并且我们在这个过程中学到了一些关于结果的有用信息。在类型层面允许数据是有用的，因为某些一致性的形式（例如事物的长度）更容易通过数据（数字）来表达。

但真正的突破是 GADT 模式匹配，其中模式的类型可以 *refine* 它匹配的参数类型。你有一个长度为  $n$  的向量；你检查它是否为空 (*nil*) 或为头 (*cons*)；现在你知道  $n$  是否为零。这是一种测试形式，其中每种情况下的代码类型比整体的类型更具体，因为在每种情况下，已经被 *learned* 的东西在类型层面上得到了反映。通过测试学习使得一种语言具有依赖类型，至少在某种程度上是如此。

这里有个傻乎乎的小游戏可以玩，不管你使用什么有类型的语言。把你的类型表达式中的每个类型变量和每个原始类型都替换为 1，然后以数值方式计算类型（把求和的相加，把乘积的相乘， $s \rightarrow t$  表示  $t$  的  $s$  次幂），看看你会得到什么：如果得到 0，你是逻辑学家；如果得到 1，你是软件工程师；如果得到 2 的幂，你是一个

电子工程师；如果你得到了无穷大，你就是一个程序员。这个游戏中发生的事情是一个粗略的尝试，旨在衡量我们所管理的信息以及我们的代码必须做出的选择。我们常用的类型系统擅长管理编码中的“软件工程”方面：解包和组合组件。但一旦做出选择，类型就无法观察到它，而一旦出现需要做出的选择，类型也无法引导我们：非依赖类型系统将给定类型中的所有值近似为相同。这对它们在防止 bug 方面的使用是一个相当严重的限制。

## 7.4 Haskell 单例：SNat 带来了什么？

我们得到什么？嗯。单例的状态是 *awkward but currently necessary workaround*，我们越早消除它们越好。

让 让我看看我是否能澄清一下这个图。我们有一个数据类型 `Na`

数据 自然数 = 零 | 后继 自然数

（战争甚至因为比 `Suc` 中 ‘`c`’ 的数量更微不足道的问题而爆发过）

`Nat` 类型具有在类型层面上无法区分的运行时值。Haskell 类型系统当前具有 *replacement* 属性，这意味着在任何类型正确的程序中，您可以将任何类型正确的子表达式替换为具有相同作用域和类型的替代子表达式，并且程序将继续保持类型正确。例如，您可以重写每个出现的

如果 `<b>` 则 `<t>` 否则 `<e>`

Translated Text: 到

如果 `<b>` 那么 `<e>` 否则 `<t>`

你可以确保，检查你的程序类型的结果不会出错。

替换性质令人尴尬。这清楚地证明，你的类型系统恰恰在语义开始变得重要的那一刻就放弃了。

现在，作为运行时值的数据类型，`Nat` 也成为了类型层次值 `'Zero` 和 `'Suc` 的一种类型。后者仅存在于 Haskell 的类型语言中，完全没有运行时的存在。请注意，尽管 `'Zero` 和 `'Suc` 存在于类型层次，但称它们为“类型”并没有帮助，那些目前这么做的人应该停止。它们没有类型 `*`，因此不能 *classify values*，而这是值得被称为类型的事物所能做到的。

运行时与类型级的 `Nats` 之间没有直接的转换手段，这可能会很麻烦。一个典型的例子涉及对 *vectors* 的一个关键操作：

数据 `Vec :: Nat -> * -> *` 其中 `VNil :: Vec 'Zero x` 和 `VCons :: x -> Vec n x -> Vec ('Suc n) x`

我们可能希望计算一个由给定元素的多个副本组成的向量（也许作为 `Applicative` 实例的一部分）。给出如下类型看起来似乎是个不错的主意

`vec :: 对于 (n :: Nat) (x :: *)。x -> Vec n x`

但是那有可能起作用吗？为了制作 `n` 份某物，我们需要在运行时知道 `n`：程序必须决定是部署 `VNil` 并停止，还是部署 `VCons` 并继续，它需要一些数据来做出这个决定。一个很好的线索是 `forall` 量词，它是 *parametric*：它表示量化的信息仅对类型可用，并在运行时被删除。

Haskell 目前强制施加了一种完全虚假的巧合，即依赖量化（forall 所做的事情）与运行时擦除之间的巧合。它确实 *not* 支持一种依赖但不被擦除的量子，我们通常称之为 *pi*。vec 的类型和实现应该类似于

```
vec :: pi (n :: Nat) -> 对于所有 (x :: *) . Vec n x
vec ' Zero x = VNil v
ec ' Suc n x = VCons x (vec n x)
```

其中位于 *pi*-位置的参数以类型语言书写，但数据在运行时是可用的。

那么我们改用什么做法呢？我们使用单例来间接地刻画成为一个 *run-time copy of type-level data* 的含义。

```
data SNat :: Nat -> * 其中 SZero :: SNat Zero
SSuc :: SNat n -> SNat (Suc n)
```

现在，SZero 和 SSuc 构成运行时数据。SNat 与 Nat 并不同构：前者的类型是 *Nat -> \**，而后者的类型是 *\**，因此试图让它们同构会产生类型错误。Nat 中有许多运行时值，而类型系统并不区分它们；在每一个不同的 SNat n 中，恰好只有一个（值得一提的）运行时值，因此类型系统不能区分它们这一事实并不是关键。关键在于，对于每个不同的 n，SNat n 都是一个不同的类型，并且 GADT 的模式匹配（其中模式可以是其所匹配的 GADT 类型的一个更具体的实例）能够细化我们对 n 的认识。

我们现在可以写出

```
vec :: forall (n :: Nat). SNat n -> forall (x :: *) . x -> Vec n x
vec SZero x = VNil
vec (SSuc n) x = VCons x (vec n x)
```

单例使我们能够通过利用唯一一种允许细化类型信息的运行时分析形式，在运行时与类型层级数据之间架起桥梁。质疑它们是否真的必要是很合理的；而目前它们之所以必要，仅仅是因为这一鸿沟尚未被消除。

## 7.5 对数据类型提升加以限制的动机

如果你推广由提升类型索引的类型，会发生一件有趣的事情。假设我们构建

```
数据 Nat = Ze | Su Nat
```

然后

```
数据 Vec :: Nat -> * -> * 其中 VNil :: Vec Ze x
VCons :: x -> Vec n x -> Vec (Su n) x
```

幕后，构造函数的 *internal* 类型表示由约束实例化的返回索引，就好像我们写了

```
data Vec (n :: Nat) (a :: *) = n `Ze` => VNil | forall k. n `Su` k => V
Cons a (Vec k a)
```

现在如果我们被允许类似这样的东西

```
数据 Elem :: forall n a. a -> Vec n a -> * 其中 Top :: Elem x (VCons x
xs) Pop :: Elem x xs -> Elem x (VCons y xs)
```

翻译成内部形式应该类似于

```
数据 Elem (x :: a) (zs :: Vec n a) = 对于所有 (k :: Nat), (xs :: Vec k a). (n `Su` k, zs `VCons` x xs) => Top | 对于所有 (k
:: Nat), (xs :: Vec k s), (y :: a). (n `Su` k, zs `VCons` y xs) => Pop (Elem x xs)
```

但是看看每种情况中的第二个约束！我们有

```
zs :: Vec n a
```

但是

```
VCons x xs, VCons y xs :: Vec (Su k) a
```

但是在当时定义的系统 FC 中，等式约束的两边必须具有相同类型，因此这个例子并非没有问题。

一种解决方法是使用第一个约束的证据来修正第二个约束，但那样我们就需要依赖约束。

```
(q1 :: n ~ Su k, zs |> q1 ~ VCons x xs)
```

另一个解决方法就是允许异构方程，正如我十五年前在依赖类型理论中所做的那样。不可避免地，会出现一些种类在方式上不明显相等的事物之间的方程。

目前更倾向于后者计划。据我了解，你们提到的政策是作为一种过渡措施，直到具有异质平等的核心语言设计（如Weirich及其同事所提议的）成熟并能够实施。我们生活在一个有趣的时代。

## 7.6 什么是索引单子？

如往常一样，人们使用的术语并不完全一致。有许多灵感来源于单子，但严格来说并不完全是单子的概念。术语“索引单子”是其中的一种（包括“单子式”和“参数化单子”（Atkey为其命名））用来描述这种概念的术语之一。（另一个类似的概念，如果你感兴趣的话，是Katsumata的“参数化效应单子”，通过一个单群进行索引，其中return是中立地索引的，bind则在其索引中累积。）

首先，让我们检查种类。

```
IxMonad (m :: 状态 -> 状态 -> * -> *)
```

也就是说，“计算”（或者如果你更喜欢“动作”，但我还是用“计算”）的类型看起来像

```
m before after value
```

其中 `before`、`after :: state`，而 `value :: *`。其思想是捕捉一种与具有某种 *predictable* 状态概念的外部系统安全交互的手段。一次计算的类型会告诉你它在运行之前状态必须是什么、运行之后状态将会是什么，以及（就像作用于 `*` 的常规模纳德一样）该计算会产生什么类型的值。

通常的零零碎碎在 `*-wise` 上像一个 `monad`，而在 `state-wise` 上则像玩多米诺骨牌。

`ireturn :: a -> m i a` -- 返回一个纯值会保持状态  
`ibind :: m i j a -> -- 我们可以从 i 到 j 并得到一个 a，继而 (a -> m j k b) -- 我们可以从 j 到 k 并得到一个 b，因此 -> m i k b -- 我们确实可以从 i 到 k 并得到一个 b`

由此生成的“Kleisli 箭头”（产生计算的函数）的概念是

`a -> m i j b` -- *a* 为输入值，*b* 为输出值；状态从 *i* 转移到 *j*

从而我们得到一个复合

`icomp :: IxMonad m => (b -> m j k c) -> (a -> m i j b) -> a -> m i k c`  
`icomp f g = \ a -> ibind (g a) f`

而且，一如期望，这些定律精确地保证 `ireturn` 和 `icomp` 给我们 一个类别

`ireturn 'icomp 'g = g f 'icomp 'ireturn = f (f 'icomp 'g`  
`) 'icomp 'h = f 'icomp '(g 'icomp 'h)`

或者，在搞笑的伪 C/Java/之类的代码中，

`g(); 跳过 = g() 跳过; f() = f() {h(); g()};`  
`f() = h(); {g(); f()};`

为什么要费心？为了对交互的“规则”建模。例如，如果光驱里没有光盘，你就无法弹出 DVD；如果光驱里已经有光盘，你也无法再放入一张 DVD。所以

`data DVDDrive :: Bool -> Bool -> * -> *` where -- `Bool` 表示“驱动器是否已满？”  
`DReturn :: a -> DVDDrive`  
`i i a DInsert :: DVD -> -- 你有一张 DVD DVDDrive True k a -> -- 你知道在已满状态下如何继续 full DVDDrive False k a -- 因此你可以在空的情况下插入 DEject :: (DVD -> -- 一旦你收到一张 DVD DVDDrive False k a) -> -- 你知道在空的情况下如何继续 empty DVDDrive True k a -- 因此你可以在已满时弹出`

`instance IxMonad DVDDrive where -- 将这些方法放到它们需要去的地方 ireturn = DReturn -- 所以这个要放到别的地方 ibind (DReturn a) k = k a ibind (DInsert dvd j) k = DInsert dvd (ibind j k) ibind (DEject j) k = DEject j $ \ dvd -> ibind (j dvd) k`

有了这一点，我们可以定义“原始”命令

`dInsert :: DVD -> DVDDrive False True ()`  
`dInsert dvd = DInsert dvd $ DReturn ()`

`dEject :: DVDDrive 真 假 DVD`  
`dEject = DEject $ \ dvd -> DReturn dvd`

从中通过`ireturn`和`ibind`组装其他部分。现在，我可以写（借用`do`符号）

```
discSwap :: DVD -> DVDDrive True True DVD discSwap dvd = do dvd' <- dEject; dI
nsert dvd ; ireturn dvd'
```

但不是物理上不可能的

```
discSwap :: DVD -> DVDDrive 真 真 DVD discSwap dvd = 执行 dInsert d
vd; dEject -- 哎呀!
```

或者，可以直接定义一个人的原始命令

```
数据 DVDCCommand :: Bool -> Bool -> * -> * 其中 InsertC :: DVD -
> DVDCCommand False True () EjectC :: DVDCCommand True False
DVD
```

然后实例化泛型模板

```
data CommandIxMonad :: (state -> state -> * -> *) -> state -> state -> * -> * 其中 C
Return :: a -> CommandIxMonad c i i a (:?) :: c i j a -> (a -> CommandIxMonad c
j k b) -> CommandIxMonad c i k b
```

实例 `IxMonad (CommandIxMonad c)` 其中

```
ireturn = CReturn
ibind (CReturn a) k = k a
ibind (c :? j) k = c :? \ a -> ibind (j a) k
```

实际上，我们已经说明了什么是原始的 Kleisli 箭头（也就是一个“多米诺骨牌”是什么），然后在其之上构建了一个合适的“计算序列”概念。

注意，对于每个索引单子 `m`，“无变化对角线”`m i i` 是一个单子，但一般来说，`m i j` 不是。此外，值没有被索引，而是计算被索引，因此，索引单子不仅仅是单子在其他范畴中的实例化。

现在，再次看看一个 Kleisli 箭头的类型

```
a -> m i j b
```

我们知道必须从状态 `i` 开始，并且我们预测任何后续都会从状态 `j` 开始。我们对这个系统了解很多！这不是一项有风险的操作！当我们把 DVD 放入光驱时，它就会进去！DVD 光驱对每条命令之后的状态没有任何决定权。

但在与世界互动时，情况一般并非如此。有时你可能需要放弃一些控制，让世界按照它自己的方式运行。例如，如果你是一个服务器，你可能会给客户端一个选择，而你的会话状态将取决于他们的选择。服务器的“提供选择”操作并不会决定结果状态，但服务器应该能够继续进行。它不像上述意义上的“原始命令”，因此索引单子不适合用来建模 *unpredictable* 场景。

有什么更好的工具？

类型 `f :-> g` = 对所有 `state`。 `f state -> g state`

```
class MonadIx (m :: (state -> *) -> (state -> *)) where
  returnIx    :: x :-> m x
  flipBindIx  :: (a :-> m b) -> (m a :-> m b)  -- tidier than bindIx
```

吓人的饼干？其实并不，因为有两个原因。第一，它看起来更像单子本身的样子，因为它 *is* 一个单子，但作用在  $(\text{state} \rightarrow *)$  上，而不是  $*$ 。第二，如果你看看一个 Kleisli 箭头的类型，

$a \rightarrow m\ b$  = 对于所有状态。a 状态  $\rightarrow m\ b$  状态

你会得到一种具有前置条件 *precondition* a 和后置条件 b 的计算类型，就像在经典的霍尔逻辑中一样。程序逻辑中的断言花了不到半个世纪，才跨越 Curry-Howard 对应关系并成为 Haskell 的类型。returnIx 的类型表明：“只要什么都不做，你就可以实现任何成立的后置条件”，这正是霍尔逻辑中关于“skip”的规则。相应的组合则是霍尔逻辑中关于“;”的规则。

让我们通过查看 bindIx 的类型来结束，把所有的量词都写出来。

$\text{bindIx} :: \text{forall } i. m\ a\ i \rightarrow (\text{forall } j. a\ j \rightarrow m\ b\ j) \rightarrow m\ b\ i$

这些全称量词具有相反的极性。我们选择初始状态 i，以及一个可以从 i 开始、后置条件为 a 的计算。世界可以选择它喜欢的任何中间状态 j，但它必须向我们提供后置条件 b 成立的证据，并且从任何这样的状态出发，我们都可以继续执行以使 b 成立。因此，按顺序，我们可以从状态 i 达成条件 b。通过放松我们对“之后”状态的约束，我们可以建模 *unpredictable* 计算。

IxMonad 和 MonadIx 都很有用。它们分别从可变状态的角度建模交互式计算的有效性，分别对应可预测和不可预测的情况。可预测性在你能够获得时非常有价值，但不可预测性有时是生活中的事实。因此，希望这个回答能对索引单子 (indexed monads) 提供一些指示，预测它们何时开始有用，何时停止有用。

## 7.7 索引集上的函子不动点

通过取一个双函子的固定点来构造递归函子是完全正确的，因为  $1 + 1 = 2$ 。列表节点结构被给定为一个具有两种子结构的容器：“元素”和“子列表”。

令人困扰的是，我们需要另一整套 Functor 的概念（尽管其名称相当一般，但它刻画的是一种相当特定的函子变体），才能将一个 Functor 构造为一个不动点。然而，我们可以（算是个小把戏）转向一种稍微更一般的函子概念，它在不动点下是 *closed*。

类型  $p \rightarrow q = \forall i. p\ i \rightarrow q\ i$

类 FunctorIx ( $f :: (i \rightarrow *) \rightarrow (o \rightarrow *)$ ) 其中  $\text{mapIx} :: (p \rightarrow q) \rightarrow f\ p \rightarrow f\ q$

这些是在 *indexed sets* 上的函子，因此这些名字并非只是对戈西尼和于德佐的随意致敬。你可以把 o 理解为“结构的种类”，把 i 理解为“子结构的种类”。这里有一个例子，基于这样一个事实： $1 + 1 = 2$ 。

数据  $\text{ListF} :: (\text{Either } () () \rightarrow *) \rightarrow (() \rightarrow *)$  其中  $\text{Nil} :: \text{ListF } p' ()$   $\text{Cons} :: p' (\text{Left } () \rightarrow p (\text{Right } () \rightarrow \text{ListF } p' ()))$

实例 FunctorIx ListF 其中

$\text{mapIx } f\ \text{Nil} = \text{Nil}$   $\text{mapIx } f\ (\text{Cons } a\ b) = \text{Cons } (f\ a)\ (f\ b)$

为了利用子结构排序的选择，我们需要一种类型层面的案例分析。我们不能仅靠一个类型函数来解决，因为

1. 我们需要它是部分应用的，但这不被允许； 2. 我们在运行时需要一点信息来告诉我们哪种排序存在。

```
data Case :: (i -> *) -> (j -> *) -> (Either i j -> *) where CaseL :: p i -> Case p q (Left i)
CaseR :: q j -> Case p q (Right j)
```

```
caseMap :: (p -> p') -> (q -> q') -> Case p q -> Case p' q'
caseMap f g (CaseL p) = CaseL (f p)
caseMap f g (CaseR q) = CaseR (g q)
```

现在我们可以取定点：

```
数据 Mu :: ((Either i j -> *) -> (j -> *)) -> ((i -> *) -> (j -> *))
其中 In :: f (Case p (Mu f p)) j -> Mu f p j
```

在每个子结构位置，我们进行案例拆分，以查看我们是应该有一个  $p$  元素还是一个  $Mu\ f\ p$  子结构。然后我们得到它的函子性。

```
实例 FunctorIx f => FunctorIx (Mu f) 其中 mapIx f (In fpr) = In (mapIx (caseMap f
(mapIx f)) fpr)
```

要从这些东西构建列表，我们需要在  $*$  和  $() -> *$  之间来回切换。

```
新类型 K a i = K {unK :: a}
```

```
类型 列表 a = Mu ListF (K a)
() 模式 NilP :: 列表 a 模式 NilP = 在 Nil 模式
ConsP :: a -> 列表 a -> 列表 a 模式 ConsP a 作为 = 在 (Cons (CaseL (K a)) (CaseR
eR 作为))
```

现在，对于列表，我们得到

```
map' :: (a -> b) -> List a -> List b
map' f = mapIx (K . f . unK)
```

## 7.8 为什么类型系统拒绝了我看似有效的程序？

如果你的程序 *really* 在你看来是成立的，那么你就应该能够在 Haskell 中写出完成你想要之事的 `get` 的类型，而不是靠空泛的手势来糊弄。让我来帮你把这些空谈具体化，并揭示你为什么会提出那种痴心妄想、想要插在棍子上的月亮的要求。

我想表达的是： $get :: (Convert\ a\ contained\ by\ D\ b) \Rightarrow D \rightarrow b$ ，这似乎是不可能的。

如所述，这并不像你需要的那么精确。实际上，这正是Haskell目前给你的，因而

```
获取 :: (转换 A b, 转换 B b, 转换 C b) => D -> b
```

任何可以被D包含的a都需要一次性转换为那个b。这就是你得到经典系统管理员逻辑的原因：除非它们都能变成b，否则不允许得到任何D。

问题在于，你需要了解的状态不是可能包含在 *any* 的旧 D 中的那种类型，而是你作为输入接收到的那个特定 D 中所包含的类型。对吧？你想要

```
打印 (获取 (DB B) :: A) -- 这个要成功    打印获取 (DC C
) :: A) -- 这个要失败
```

但是 DB B 和 DC C 只是 D 的两个不同元素，而就 Haskell 的类型系统而言，在每个类型之内，所有不同的东西都是相同的。如果你想区分 D 的各个元素，那么你需要一个 D-依赖类型。下面是我随手勾勒的写法。

```
晚餐 :: D -> * 晚餐 (DA
a) = A 晚餐 (DB b) = B
晚餐 (DC c) = C
```

```
获取 :: 对于 x. pi (d :: D) -> (转换 (DInner d) x) => x
获取 (DA x) = 转换 x 获取 (DB
x) = 转换 x 获取 (DC x) = 转换
x
```

其中 pi 是用于传递在运行时的数据的绑定形式（不同于 forall），但类型可能依赖于此（不同于 ->）。现在，约束不是在讨论任意的 Ds，而是讨论你手中的那个 d :: D，并且约束可以通过检查其 DInner 来精确计算所需的内容。

没有什么你能说的能让它消失，只有我的 pi。

遗憾的是，尽管  $\pi$  正在迅速从天而降，但它还没有着陆。尽管如此，与月亮不同，它可以用棍子到达。毫无疑问，你会抱怨我改变了设置，但实际上我只是将你的程序从2017年的Haskell翻译成2015年的Haskell。总有一天，你会把它拿回来，带着我随意挥手的那个类型。

你无话可说，但你可以 sing。

步骤 1. 打开 DataKinds 和 KindSignatures，并为您的类型构建单例（或者让 Richard Eisenberg 为您做这件事）。

```
data A = A deriving Show data Aey :: A -> * where -- 把“-ey”想作形容词后缀 Aey :: Aey ' A
-- 就像“tomatoey”一样
```

```
数据 B = B 派生 Show 数据 Bey ::
B -> * 其中 Bey :: Bey ' B
```

```
数据 C = C 导出 Show 数据 Cey ::
C -> * 其中 Cey :: Cey ' C
```

```
数据 D = DA A | DB B | DC C 派生 Show
数据 Dey :: D -> * 其中 DAey :: Aey a ->
Dey (DA a) DBey :: Bey b -> Dey (DB b)
DCey :: Cey c -> Dey (DC c)
```



```

获取 :: 对于 x. 对于 d. Dey d -> (转换 (DInner d) x) => x
获取 (DAey x) = 转换 (sung x) 获取 (DBey x)
) = 转换 (sung x) 获取 (DCey x) = 转换 (sung x)

```

放心，当我们有了正确的 `pi` 时，那些 `DAeys` 将会是真正的 `DAs`，而那些 `xs` 将不再需要被唱出来。我的 `get` 手势类型将是 `Haskell`，而你的 `get` 代码将没问题。但在此期间

```

main = 执行 打印 (获取 (DCey Cey) :: B)
打印 (获取 (DBey Bey) :: A)

```

完全可以通过类型检查。也就是说，你的程序（加上 `DInner` 以及 `get` 的正确类型）看起来像是合法的 `Dependent Haskell`，我们已经快完成了。

## 7.9 在 Haskell 中编写和检查不变量是否可行？

以下是一个特技，但它是相当安全的技术，所以可以在家尝试。它使用了一些有趣的新玩具，将 `order` 不变式融入 `mergeSort`。

```

{-# LANGUAGE GADTs, PolyKinds, KindSignatures, MultiParamTypeClasses, FlexibleInstances, RankNTypes, FlexibleContexts #-}

```

我将使用自然数，只为保持简单。

```

数据 Nat = Z | S Nat 派生 (Show, Eq, Ord)

```

但我会类型类 `Prolog` 中定义 `<=`，这样类型检查器就可以尝试隐式地推断出顺序。

```

class LeN (m :: Nat) (n :: Nat) where instance LeN Z n where i
nstance LeN m n => LeN (S m) (S n) where

```

为了排序数字，我需要知道任何两个数字都可以排序 *one way or the other*。我们来定义两个数字可以排序的含义。

```

数据 OWOTO :: Nat -> Nat -> * 其中 LE :: LeN x
y => OWOTO x y GE :: LeN y x => OWOTO x y

```

我们希望确认，只要我们有它们的运行时表示，任意两个数确实都是可比较的。如今，我们通过为 `Nat` 构建 *singleton family* 来做到这一点。`Natty n` 是 `n` 的运行时副本的类型。

```

数据 Natty :: Nat -> * 其中 Zy :: Natty Z Sy
:: Natty n -> Natty (S n)

```

测试数字的方向与通常的布尔版本非常相似，只是加入了证据。步进情况需要进行拆包和重打包，因为类型发生了变化。实例推断对于所涉及的逻辑非常有用。

```

owoto :: 对于 m n. Natty m -> Natty n -> OWOTO m n owoto Zy n = LE ow
oto (Sy m) Zy = GE owoto (Sy m) (Sy n) = 案例 owoto m n 为 LE -> LE GE
-> GE

```

现在我们知道如何将数字排序，让我们看看如何制作有序列表。计划是描述如何保持顺序 *between loose bounds*。当然，我们不希望排除任何元素使其不可排序，因此 *bounds* 的类型通过底部和顶部元素扩展了元素类型。

数据绑定  $x = \text{机器人} \mid \text{值 } x \mid \text{顶部派生 (显示, 等于, 顺序)}$

我相应地扩展了  $\leq$  的概念，以便类型检查器可以进行范围检查。

```
class LeB (a :: Bound Nat) (b :: Bound Nat) where
instance      LeB Bot      b      where
instance LeN x y => LeB (Val x) (Val y) where
instance      LeB (Val x) Top      where
instance      LeB Top      Top      where
```

这是数字的有序列表：一个  $\text{OList } l \ u$  是一个序列  $x_1 :< x_2 :< \dots :< x_n :< \text{ONil}$ ，其中  $l \leq x_1 \leq x_2 \leq \dots \leq x_n \leq u$ 。  $x :<$  检查  $x$  是否高于下界，然后将  $x$  作为尾部的下界。

数据  $\text{OList} :: \text{Bound Nat} \rightarrow \text{Bound Nat} \rightarrow *$  其中  $\text{ONil} :: \text{LeB } l \ u \Rightarrow \text{OList } l \ u$   
 $\text{ist } l \ u (:<) :: \text{forall } l \ x \ u. \text{LeB } l \ (\text{Val } x) \Rightarrow \text{Natty } x \rightarrow \text{OList } (\text{Val } x) \ u \rightarrow \text{OList } l \ u$

我们可以像处理普通列表一样为有序列表编写 *merge*。关键不变式在于：如果两个列表共享相同的边界，那么它们的合并结果也共享相同的边界。

合并  $:: \text{OList } l \ u \rightarrow \text{OList } l \ u \rightarrow \text{OList } l \ u$  合并  $\text{ONil } lu = lu$  合并  $lu$   
 $\text{ONil} = lu$  合并  $(x :< xu) (y :< yu) = \text{情况 } \text{owoto } x \ y \text{ 为 } \text{LE} \rightarrow x :<$   
 合并  $xu (y :< yu) \text{GE} \rightarrow y :< \text{合并 } (x :< xu) \ yu$

案件分析的分支扩展了从输入中已知的内容，提供了足够的排序信息以满足结果的要求。实例推理充当基本的定理证明者：幸运的是（或者说，通过一些练习），证明的义务足够简单。

让我们达成协议。我们需要为数字构造运行时证明，以便按照这种方式对它们进行排序。

数据  $\text{NATTY} :: *$  其中  $\text{Nat} :: \text{Natty}$   
 $n \rightarrow \text{NATTY}$

$\text{natty} :: \text{Nat} \rightarrow \text{NATTY}$   $\text{natty } Z = \text{Nat } Zy \text{ natty } (S \ n) = \text{case natty } n \text{ of } \text{Nat } n \rightarrow \text{Nat } (Sy \ n)$

我们需要相信这个翻译能给我们对应于我们想排序的  $\text{Nat}$  的  $\text{NATTY}$ 。 $\text{Nat}$ 、 $\text{Natty}$  和  $\text{NATTY}$  之间的相互作用有点让人沮丧，但这正是当前 *Haskell* 中的做法。一旦我们得到了这一点，我们就可以像往常一样以分治法构建排序。

处理  $:: [x] \rightarrow ([x], [x])$  处理  $[] = ([], [])$  处理  $(x : xs) = (x : zs, ys)$  其中  $(ys, zs) = \text{处理 } xs$

排序 :: [Nat] -> OList Bot Top 排序 [] = ONil 排序 [n] = case natty n of Nat n -> n :< ONil  
 排序 xs = 合并 (排序 ys) (排序 zs) 其中 (ys, zs) = 处理 xs

我常常感到惊讶，许多对我们有意义的程序同样能对类型检查器有意义。

这是我做的一些备用工具，用来看看发生了什么。

```
``markdowninstance Show (Natty n) where show Zy = "Zy" show (
Sy n) = "(Sy " ++ show n ++ ")" instance Show (OList l u) where sh
ow ONil = "ONil" show (x :< xs) = show x ++ " :< " ++ show xs ni
:: Int -> Nat ni 0 = Z ni x = S (ni (x - 1))``
```

并且没有什么隐藏的。

## 7.10 为什么类型转换是一件坏事？

人们认为在类型上进行模式匹配是错误的，这真的很奇怪。我们在数据上进行模式匹配时得到了很多好处，尤其是当我们进行宇宙构造时，这些数据具有`encode`类型。如果你采用我和Thorsten Altenkirch开创的方法（以及我和我的战友开始设计的方法），类型确实形成了一个`closed`宇宙，因此你甚至不需要解决（坦率地说值得解决的）使用开放数据类型进行计算的问题，以将类型视为数据。如果我们能够直接对类型进行模式匹配，我们就不需要解码函数将类型代码映射到其含义，最坏的情况是减少了杂乱无章，最好的是减少了通过等式法则来证明和强制解码函数行为的需求。我完全打算通过这种方式构建一个无中介的封闭类型理论。当然，你需要那个0级类型占据1级数据类型。这在你构建归纳递归宇宙层次结构时自然而然地发生。

但是，关于参数化性，我听到你在问？

首先，当我试图编写类型泛化代码时，我不希望有参数化约束。不要强制我使用参数化。

其次，为什么类型应该是我们唯一的参数化对象？为什么我们不可以其他东西上进行参数化，比如完全普通的类型索引，它们存在于数据类型中，但我们在运行时不出现？仅仅因为它们的类型，必须存在的量，这在`specification`中起作用，实在是非常麻烦。

域的类型与是否应对其进行参数化量化有`nothing whatsoever`关系。

让我们设定（例如，正如Bernardy及其团队所提议的）一种学科，其中参数化/可擦除和非参数化/可匹配的量化是明确区分并且都可以使用的。那么类型可以是数据，我们仍然可以表达我们的意思。

## 7.11 为什么我不能对类型族进行模式匹配？

当你声明

```
leftthing :: a -> Leftthing a
```

你是在说 `leftthing` 的 *caller* 可以选择 `a` 是什么。当你随后写下

左物 (两物 `a b`) = 左物 `a`

你 *presuming* 他们已经选择了一个 `Twothings` 类型，而鉴于这并不一定是事实，你的程序被拒绝。

你可能认为你是 *testing whether*，他们选择了 `Twothings` 类型，但不是！类型信息在运行时之前被擦除，因此无法进行这样的测试。

你 *can* 尝试恢复必要的运行时信息。首先让我修正你在 `Leftthing` 和 `leftthing` 之间的不一致。

类型族 `Leftthing a` 其中

左事物 (两个事物 `a b`) = 左事物 { -你忘了递归! - } `a` 左事物 `a = a`

现在我们可以定义关于 `Twothingness` 的见证的 GADT。

数据 `IsItTwothings :: * -> *` 其中

是的，它是 `::` 它是两件事 `a ->` 它是两件事 (两件事 `a b`) 不是的 `::` 左边的东西 `a ~a =>` 它是两件事 `a --` ~~~~~ 这个约束将适用于任何类型 -- 这 \*绝对不是\* 一个两件事类型

然后我们可以将证人作为一个参数传递：

`leftthing ::` 它是两个事物吗 `a -> a ->` 左事物 `a leftthing` (是的，它是 `r`) (两个事物 `a b`) = `leftthing r a leftthing` 不，它不是 `b = b`

实际上，该见证就是你类型根部左嵌套的 `Twothingses` 数量的一元编码。这些信息足以在运行时确定需要进行正确解包次数。

> 左侧物 (是的它是 (是的它是 不是它不是)) (两个物 (两个物 真 11) (两个物  
True

总而言之，你不能通过对一个值进行模式匹配来找出它的类型。相反，你需要先知道类型才能进行模式匹配（因为类型决定了内存布局，而且在运行时没有类型标签）。你也不能直接对类型进行模式匹配（因为它们根本不存在，无法被匹配）。你可以构造一些数据类型，使其在运行时充当类型结构的证据，然后对这些证据进行匹配。

也许，某一天，如果你给它类型，你的程序就能正常工作

左物 `::`  $\pi$  `a. a ->` 左物 `a`

其中  $\pi$  是依赖量词，表示隐藏类型参数不会被删除，而是在运行时传递并匹配。那一天尚未到来，但我认为它会到来。

## 7.12 正整数类型

如果我不顺便推荐一下 Adam Gundry 的 `Inch` 预处理器——它用于为 Haskell 管理整数约束——那我作为他的导师就失职了。

智能构造函数和抽象屏障都很好，但它们将太多的测试推到运行时，并且没有考虑到你可能实际上知道自己在做什么，并且以一种静态检查无误的方式进行，没有必要使用 `Maybe` 填充。（一个挑剔的写手。作者）

另一则回答的内容似乎暗示 0 是正的，这一点一些人可能会认为有争议。当然，事实是我们对各种下界都有用途，0 和 1 都经常出现。我们对上界也有一些用途。)

在习的DML传统中，Adam的预处理器在Haskell原生提供的基础上增加了额外的精度，但生成的代码仍然保持Haskell原样。若能将他所做的工作与GHC更好地集成，并与Iavor Diatchki在类型层级自然数方面的工作协调起来，那就太好了。我们非常希望弄清楚有哪些可能性。

回到一般性的观点，目前 Haskell 还不具备足够的依赖类型能力，无法通过概括来构造子类型（例如，大于 0 的 Integer 元素），但你通常可以将类型重构为一种更具索引性的版本，从而允许静态约束。目前，*singleton* 类型构造是现有这些不太理想的方法中最为整洁的一种来实现这一点。你需要一个由“静态”整数构成的 *kind*，然后，种类为 `Integer -> *` 的居留者能够刻画特定整数的性质，例如“具有动态表示”（这就是单例构造，为每个静态事物赋予一个唯一的动态对应物），以及更为具体的性质，比如“为正数”。

*Inch* 表示一种设想：如果在处理一些行为相当良好的整数子集时，你不必为了工作而费心使用单例 (*singleton*) 构造，那会是什么样子。依赖类型编程在 Haskell 中往往是可行的，但目前却比必要的要复杂。对这种情况最恰当的情绪是尴尬，而我个人对此感受尤为强烈。

## 7.13 测试一个值是否匹配构造器

带标签联合的标签本应是一等值，而且只要稍费周折，它们确实可以做到。花招预警：

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures, TypeFamilies,
PolyKinds, FlexibleInstances, PatternSynonyms #-}
```

步骤一：定义标签的类型级版本。

数据 标签类型 = 空标签 | 单一标签 | 配对标签 | 多标签

第二步：定义类型级标签可表示性的值级见证。Richard Eisenberg 的 *Singletons* 库可以为你完成这项工作。我的意思是类似这样的：

```
数据标签 :: 标签类型 -> * 其中 空T :: 标
签 空标签 单T :: 标签 单标签 对T :: 标签
对标签 多T :: 标签 多标签
```

现在我们可以说明，对于给定的标签，我们期望找到哪些与之相关的内容。

```
类型族 Stuff (t :: TagType) :: * 其中 Stuff EmptyTag = () Stu
ff SingleTag = Int Stuff PairTag = (Int, Int) Stuff LotsTag = [I
nt]
```

因此我们可以重构你最初想到的类型

```
数据 NumCol :: * 其中 (:&) :: Tag t -> Stuff t -> Nu
mCol
```

并使用 `PatternSynonyms` 恢复您所期望的行为：

模式 `Empty = EmptyT :& ()` 模式 `Single i = SingleT :& i` 模式 `Pair i j = PairT :& (i, j)` 模式 `Lots is = LotsT :& is`

所以发生的情况是，每个 `NumCol` 的构造函数已经变成了一个由它所对应的标签类型索引的标签。也就是说，构造函数标签现在与其他数据分开存储，通过一个共同的索引来同步，确保与标签相关的内容与标签本身相匹配。

但我们可以只讨论标签。

数据 `Ex :: (k -> *) -> *` 其中 -- 希望在这里说 `newtype Witness :: p x -> Ex p`

现在，`Ex Tag` 是“具有类型层次对应物的运行时标签”的类型。它具有 `Eq` 实例

实例 `Eq (Ex Tag)` 其中

见证 `EmptyT ==` 见证 `EmptyT = 真` 见证 `SingleT ==` 见证 `Single T = 真` 见证 `PairT ==` 见证 `PairT = 真` 见证 `LotsT ==` 见证 `LotsT = 真` `_ == _ = 假`

此外，我们可以轻松提取 `NumCol` 的标签。

`numColTag :: NumCol -> Ex 标签 numCol`  
`Tag (n :& _) = 见证 n`

这使我们能够匹配您的规格。

`filter ((Witness PairT ==) . numColTag) :: [NumCol] -> [NumCol]`

这就引出了一个问题：你的规范是否真的就是你所需要的。关键在于，检测到一个标签会赋予你对该标签所包含内容的期望。输出类型 `[NumCol]` 并不能充分体现这样一个事实：你知道自己刚刚得到的只是那些成对的数据。

你可以如何在仍然交付的同时收紧你函数的类型？

## 7.14 Haskell 和 Idris 的区别：类型宇宙中运行时/编译时的反映

是的，你观察到 `Idris` 中的类型与值的区分并不符合仅在编译时与在编译时和运行时都存在的区分，这是正确的。这其实是一个好事。拥有仅在编译时存在的值是有用的，就像在程序逻辑中我们仅在规范中使用的“幽灵变量”一样。能够在运行时拥有类型表示也是有用的，这样可以支持数据类型的泛型编程。

在 `Haskell` 中，`DataKinds`（和 `PolyKinds`）让我们写

类型族 `Cond (b :: Bool)(t :: k)(e :: k) :: k` where `Cond ' True t e = t` `Cond ' False t e = e`

并且在不太遥远的将来，我们将能够写

`e`

```
item :: pi (b :: Bool) -> Cond b Int [Int]
item True = 42
item False = [1,2,3]
```

但在该技术实现之前，我们必须将就使用依赖函数类型的单例伪造，例如这样：

```
数据 Booly :: Bool -> * 其中 Truey :: Bool
ly ' True Falsey :: Booly ' False
```

```
item :: forall b. Booly b -> Cond b Int [Int]
item Truey = 42
item Falsey = [1,2,3]
```

靠这种造假你也能走得相当远，但如果我们拥有真正的东西，一切都会容易得多。

至关重要的是，Haskell 的计划是保持并区分 `forall` 和 `pi`，分别支持参数化多态和特设多态。与 `forall` 配套的 `lambda` 和应用在运行时代码生成时仍然可以被擦除，就像现在一样，但与 `pi` 配套的则会被保留。引入运行时类型抽象 `pi x :: * -> ...` 也是合理的，并且可以把 `Data.Typeable` 那一团乱麻般的复杂性直接扔进垃圾桶。

## 7.15 为什么 GADT/存在性数据构造器不能在懒惰模式中使用？

考虑

```
数据 EQ a b 其中 Refl ::
EQ a a
```

如果我们能够定义

```
传递 :: Eq a b -> a -> b
传递 Refl a = a
```

然后我们可以有

```
运输未定义 :: a -> b
```

从而获得

```
运输未定义 真 = 真 :: 整数 -> 整数
```

然后是崩溃，而不是在尝试对未定义进行头部归一化时得到的更优雅的成功。

GADT 数据表示证据 *about* 类型，因此伪造的 GADT 数据会威胁类型安全。为了验证这些证据，有必要对它们采取严格性：在存在 `bottom` 的情况下，不能信任未求值的计算。

## 7.16 GADTs 能否用于在 GHC 中证明类型不等式？

这是Philip JF解法的简短版本，也是依赖类型理论家多年来一直用来反驳等式的方法。

类型族 Discriminate x 类型实例 Discriminate Int = ()

类型实例 Discriminate Char = Void

传递 :: 相等 a b -> 判别 a -> 判别 b 传递 自反 d = d

反驳 :: 等于 Int Char -> Void 反驳 q = 传递 q  
()

为了表明事物是不同的，你必须通过提供一个计算上下文来捕捉它们 *behaving differently*，从而产生不同的观察结果。Discriminate 正是提供了这样的上下文：一个类型级程序，它以不同的方式处理这两种类型。

不必诉诸于undefined来解决这个问题。完全编程有时涉及拒绝不可能的输入。即使undefined可用，我也建议在完全方法足够的情况下不要使用它：完全方法*explains*解释某事为何不可能，并且类型检查器确认；undefined仅仅记录*your promise*。实际上，这种反驳方法就是Epigram处理“不可能情况”的方式，同时确保案例分析覆盖其域。

就计算行为而言，注意到通过传输进行的 refute 必然在 q 上是严格的，而且 q 在空上下文中无法计算到头部正常形式，这很简单，因为不存在这样的头部正常形式（当然也因为计算会保持类型）。在一个全函数的设定中，我们可以确信 refute 永远不会在运行时被调用。在 Haskell 中，我们至少可以确定，在我们被迫对其作出响应之前，它的参数要么会发散，要么会抛出一个异常。一个 lazy 版本，例如

荒谬等式 e = 错误 "你可能有一个类型错误" 导致大问题

会忽略 e 的毒性并告诉你在没有错误时你有类型错误。我更喜欢

荒谬的相等 e = e 'seq ' 错误 "如果发生这种情况，就告我"

如果诚实的反驳太像艰苦的工作。

## 7.17 实现长度索引列表的拉链

###对齐###

依赖类型编程就像做两个拼图，而某个恶作剧者将它们粘在了一起。少一些比喻地说，我们在值层面和类型层面表达同时进行的计算，并且我们必须确保它们的兼容性。当然，我们每个人都是自己的恶作剧者，所以如果我们能够将拼图对齐，我们就会更容易完成它。当你看到类型修复的证明义务时，你可能会忍不住问

我需要添加某种证明对象（数据 Refl a b，其中 Refl :: Refl a a 等）吗，还是有某种方法只通过添加更明确的类型签名就能实现这个功能？

但是你或许可以先考虑，值层级与类型层级的计算在哪些方面不对齐，以及是否有希望让它们更接近。

###解决方案###

这里的问题是如何从一个向量中计算出选择的向量（按长度索引的列表）。因此我们希望得到一个具有如下类型的东西

List (Succ n) a -> List (Succ n) (a, List n a)

其中每个输入位置上的元素都会用其兄弟元素构成的、长度少一的向量进行装饰。所提出的方法是从左到右扫描，将较年长的兄弟累积到一个在右侧增长的列表中，然后在每个位置与较年轻的兄弟进行拼接。在右侧增长列表总是令人担忧，尤其当长度的 Succ 与左侧的 Cons 对齐时。对拼接的需求迫使引入类型级加法，但由右端活动产生的算术与加法的计算规则并不对齐。我稍后会回到这种风格，不过我们先再试着重新思考一遍。

在我们深入任何基于累加器的解决方案之前，先尝试一下最普通的结构递归。我们有“one”的情况和“more”的情况。

```
picks (Cons x xs@Nil) = Cons (x, xs) Nil
picks (Cons x xs@(Cons _ _)) = Cons (x, xs) (undefined (picks xs))
```

在这两种情况下，我们将第一个分解放在最前面。在第二种情况下，我们已经检查过尾部是非空的，因此我们可以要求其选择。我们有

```
x :: a
xs :: List (Succ n) a
picks xs :: List (Succ n) (a, List n a)
```

我们想要

```
Cons (x, xs) (undefined (picks xs)) :: List (Succ (Succ n)) (a, List (Succ n) a)
undefined (picks xs) :: List (Succ n) (a, List (Succ n) a)
```

因此，这个 undefined 需要是一个函数，它通过将 x 重新接到左端来扩展所有同级列表（而左端性是好的）。因此，我为 List n 定义了 Functor 实例。

```
实例 Functor (List n) 其中 fmap f Nil = Nil
fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

而我诅咒前奏和

```
import Control.Arrow((**))
```

这样我就可以写

```
picks (Cons x xs@Nil) = Cons (x, xs) Nil
picks (Cons x xs@(Cons _ _)) = Cons (x, xs) (fmap (id *** Cons x) (picks xs))
```

这就完成了任务，丝毫没有任何补充，更不用说给出相关的证明了。

###变体### 我感到恼火的

在两行里做同样的事情，所以我试着扭动 o

不在状态：

```
picks :: m `Succ` n => List m a -> List m (a, List n a) -- 无法通过类型检查
picks Nil = Nil
picks (Cons x xs) = Cons (x, xs) (fmap (id *** (Cons x)) (picks xs))
```

但是 GHC 会积极地求解该约束，并拒绝将 Nil 作为一个模式。而且这样做是正确的：在我们在静态上已经知道 Zero `Succ` n 的情况下，我们确实不应该进行计算，因为我们很容易构造出会发生段错误的东西。问题只是我把约束放在了一个作用域过于全局的位置。

相反，我可以为结果类型声明一个包装器。

数据 `Pick :: 自然数 -> * -> *` 其中

选择 `:: {unpick :: (a, List n a)} -> 选择 (Succ n) a`

成功 `n` 返回指数意味着非空约束是 *local* 对于一个选择。一个辅助函数执行左端扩展,

```
pCons :: a -> Pick n a -> Pick (Succ n) a
pCons b (Pick (a, as)) =
  Pick (a, Cons b as)
```

留下我们与

```
picks' :: List m a -> List m (Pick m a)
picks' Nil = Nil
picks' (Cons x xs) = Cons (Pick (x, xs)) (fmap
  ap (pCons x) (picks' xs))
```

如果我们想要

挑选 = `fmap 反挑选 . 选择'`

这也许有些过度, 但如果我们想把年长和年幼的兄弟姐妹区分开来, 把列表分成三份, 可能还是值得的, 比如这样:

数据 `Pick3 :: Nat -> * -> *` 其中 `Pick3 :: List m a -> a -> List n a -> Pick3 (Succ (m + n)) a`

```
pCons3 :: a -> Pick3 n a -> Pick3 (Succ n) a
pCons3 b (Pick3 bs x as) =
  Pick3 (Cons b bs) x as
```

```
picks3 :: List m a -> List m (Pick3 m a)
picks3 Nil = Nil
picks3 (Cons x xs) = Cons (Pick3 Nil x xs) (fmap (p
  Cons3 x) (picks3 xs))
```

再次, 所有的动作都是左端的, 因此我们与+的计算行为非常契合。

###累积###

如果我们想保持原始尝试的风格, 随着进行逐步累积“前辈兄弟”, 我们不妨将它们保持在 *zipper-style*, 将最接近的元素存储在最易访问的位置。也就是说, 我们可以逆序存储“前辈兄弟”, 这样在每一步我们只需要使用 `Cons`, 而不是连接。当我们想在每个位置构建完整的兄弟列表时, 我们需要使用逆向连接 (实际上, 是将一个子列表插入到列表拉链中)。如果你部署了 *abacus-style* 的加法, 可以轻松为向量输入 `revCat`。

类型族 `(+/) (a :: Nat) (b :: Nat) :: Nat` 类型实例 `(+/) Zero n = n` 类型实例 `(+/) (Succ m) n = m +/ Succ n`

这是与 `revCat` 中的值级计算一致的加法, 定义如下:

```
revCat :: List m a -> List n a -> List (m +/ n) a
revCat Nil ys = ys
revCat (Cons x xs) ys = revCat xs (Cons x ys)
```

我们获取了一个压缩版的 `go` 版本

`picksr :: List (Succ n) a -> List (Succ n) (a, List n a)` `picksr = go Nil` 其中

```
go :: List p a -> List (Succ q) a -> List (Succ q) (a, List (p +/ q) a)
go p (Cons x xs@Nil) = Cons (x, revCat p xs) Nil
go p (Cons x xs@(Cons _ _)) = Cons (x, revCat p xs) (go (Cons x p) xs)
```

而且没有人证明任何事情。###结论#  
## 莱奥波德·克罗内克本该说

上帝创造了自然数 *to perplex us*: 其余的都是人类的工作。

一个 `Succ` 看起来很像另一个，所以很容易写出表达式，这些表达式以与其结构不对齐的方式给出事物的大小。当然，我们可以并且应该（并且即将）为GHC的约束求解器配备改进的类型级数值推理工具。但在这之前，值得做的事情是策划将 `Conses` 与 `Succs` 对齐。

## 7.18 整数模运算的单群

在我评论的基础上，这是一个初步的尝试。模是由类型强制的，但不是代表的标准选择：那只是通过计算完成的，因此需要一个抽象层次。也有有界数的类型，但它们需要更多的工作。

进入，{`-# 语言厨房水槽 #-`}。我的意思是（实际上还不错）

```
{-# LANGUAGE DataKinds, GADTs, KindSignatures, FlexibleInstances #-}
```

让我们开始吧。首先，出于反射性，我介绍哈索基自然数：

```
数据 Nat = Z | S Nat -- 类型级数字
数据 Natty :: Nat -> * 其中 -- Nat 的值级表示
Zy :: Natty Z
Sy :: Natty S n
Natty n -> Natty (S n) 类 NATTY n 其中 -- 值级可表示性
natty :: Natty n 实例 NATTY Z 其中 natty = Zy
实例 NATTY n => NATTY (S n) 其中 natty = Sy natty
```

在我看来，这正是当你想声明一个数据类型，并允许其他类型依赖于其值时所做的事情。Richard Eisenberg 的“单例”库自动化了构造过程。

如果例子继续使用数字来索引向量，有些人指出 `()` 中的向量也可以作为 `Nat` 的单例。他们在技术上是正确的，当然，但这有些误导。当我们将 `Natty` 和 `NATTY` 视为从 `Nat` 系统生成时，它们是我们可以根据需要利用或不利用的权利，而不是额外的东西来证明这个概念。这个例子并不涉及向量，引入向量仅仅是为了为 `Nat` 提供单例将是荒谬的。

我手工写了一堆转换函数和 `Show` 实例，这样不说别的，至少能让我们看清自己在做什么。

```
int :: Nat -> 整数
int Z = 0
int (S n) = 1 + int n
```

实例 Show Nat, 其中 show = show . int

```
nat :: Natty n -> Nat nat Zy = Z
nat (Sy n) = S (nat n)
```

实例 Show (Natty n) where show = show . nat

现在我们已经准备好声明 Mod。

数据 Mod :: Nat -> \* 其中 (:%) :: 整数 -> Natty n -> Mod (S n)

类型携带模量。数值携带等价类的未归一化代表，但我们最好弄清楚如何将其归一化。对一元数的除法是一项我小时候学会的独特运动。

余数 :: 自然数 n -- 模数的前驱 -> 整数 -- 任意代表 -> 整数 -- 规范代表

```
-- 如果候选为负数，添加模余数 n x | x < 0 = 余数 n (int (nat (Sy n)) + x) -- 否则获取除法余数 n x = go (Sy n) x x 其中 go :: Natty m -- 除数倒计时 (最初是模数) -> Integer -- 我们当前对代表值的猜测 -> Integer -- 被除数倒计时 -> Integer -- 标准代表值 -- 当被除数用尽时，猜测的代表值即为标准值 go _ c 0 = c -- 当除数用尽但被除数尚未用尽时， -- 当前被除数倒计时是更好的代表值猜测， -- 但可能仍然太大，因此重新开始， -- 从模数倒计时 (方便地仍然在作用域内) go Zy _ y = go (Sy n) y y -- 否则，两个倒计时都减一 go (Sy m) c y = go m c (y - 1)
```

现在我们可以创建一个智能构造器。

```
rep :: NATTY n -- 我们从空气中提取模数 rep => 整数 -> Mod (S n) -- 当我们看到模数时，我们希望 rep x = 余数 n x :% n 其中 n = natty
```

然后 Monoid 实例就很容易了：

```
实例 NATTY n => 单群 (模 (S n)) 其中 mempty = rep 0 mappe
nd (x :% _) (y :% _) = rep (x + y)
```

我还扔了一些其他东西：

实例 `Show (Mod n)`, 其中 `show (x :: % n) = 连接 ["(", 显示 (remainder n x), ":% ", 显示 (Sy n), ")"]` 实例 `Eq (Mod n)`, 其中

```
(x :: % n) == (y :: % _) = 余数 n x == 余数 n y
```

稍微方便一点……

类型 `四 = S (S (S (S Z)))`

我们得到

```
> foldMap rep [1..5] :: Mod Four (3 :: % 4)
```

所以是的, 你确实需要依赖类型, 但 `Haskell` 已经足够依赖类型化了。

## 7.19 是否存在看起来不像容器的非平凡 `Foldable` 或 `Traversable` 实例?

每一个有效的 `Traversable f` 是同构于 `Normal s` 对于某些 `s :: Nat` > \* 在哪里

数据 `正常 (s :: Nat -> *) (x :: *)` 其中 `-- 正常是吉拉尔术语 (:-) :: s n -> Vec n x -> 正常 s x`

数据 `Nat = 零 | 继 Nat`

```
data Vec (n :: Nat) (x :: *) where Nil :: Vec Zero n (::) :: x -
```

```
> Vec n x -> Vec (Suc n) x
```

但在 `Haskell` 中实现 `iso` 并非 `trivial` (但值得尝试使用完整的依赖类型)。从道义上讲, 你选择的 `s` 是

数据 `{- 其实不是 -} 形状大小 (f :: * -> *) (n :: Nat) 其中 尺寸 :: pi (xs :: f ()) -> 形状大小 f (length xs)`

并且同构的两个方向分别分离和重组形状与内容。一个事物的形状仅由 `fmap (const ())` 给出, 关键点是, `f x` 的形状长度等于 `f x` 本身的长度。

向量是可以按从左到右的顺序遍历的。法线可以通过保持形状 (因此也保持大小) 并遍历元素的向量来准确遍历。可遍历性意味着具有有限数量的元素位置, 按线性顺序排列: 与一个正常的函子同构, 正好揭示了元素的线性顺序。相应地, 每个可遍历结构都是一个 (有限的) 容器: 它们具有一组形状和大小, 并且通过自然数的初始段 (严格小于大小) 给出了相应的位置概念。

折叠 (`Foldable`) 事物也是有限的, 并且它们保持事物的顺序 (有一个合理的 `toList`), 但它们不一定是函子 (`Functor`), 因此它们没有像 *shape* 这样的明确概念。从这个意义上讲 (即我同事 `Abbott`、`Altenkirch` 和 `Ghani` 定义的“容器”意义), 它们不一定可以用形状和位置的特征来描述, 因此不是容器。如果运气好, 其中一些可能是容器, 直到某些商的存在。事实上, `Foldable` 的存在是为了允许处理像 `Set` 这样的结构, 它们的内部结构旨在保密, 且显然依赖于元素的排序信息, 而这个信息不一定被遍历操作所遵守。然而, 究竟什么构成一个行为良好的 `Foldable` 是一个有争议的问题: 我不打算与该库设计选择的务实好处争论, 但我希望有一个更清晰的规范。

## 7.20 GHC 的类型家族是 System F-omega 的一个例子吗？

System F-omega 在 *higher kinds* 上允许全称量化、抽象和应用，因此不仅作用于类型（位于种类  $*$ ），也作用于种类  $k_1 \rightarrow k_2$ ，其中  $k_1$  和  $k_2$  本身是由  $*$  和  $\rightarrow$  生成的种类。因此，类型层本身成为一个简单类型  $\lambda$ -演算。

Haskell 提供的功能稍逊于 F-omega，因为其类型系统允许在更高的种类上进行量化和应用，但不支持抽象。在更高种类上进行量化是我们能够拥有如下类型的原因：

`fmap :: 对于所有 f, s, t. 函子 f => (s -> t) -> f s -> f t`

带有 `f :: * -> *`。相应地，像 `f` 这样的变量可以通过更高阶类型表达式实例化，例如 `Either String`。缺乏抽象使得可以通过标准的一级技术解决类型表达式中的统一问题，这些技术构成了 Hindley-Milner 类型系统的基础。

然而，*type families* 并不是真正引入高阶类型的另一种方式，也不是缺失的类型级 `lambda` 的替代品。关键是，它们必须是 *fully applied*。所以你的例子，

类型族 `Foo a` 类型实例 `Foo Int = Int` 类型  
实例 `Foo Float = ... ..`

不应视为引入某些

`Foo :: * -> * -- 这不是发生的事情`

因为 `Foo` 本身并不是一个有意义的类型表达式。我们只有一个更弱的规则：只要 `t :: *`，就有 `Foo t :: *`。

然而，类型族确实作为一种超越 F- $\omega$  的独立类型层级计算机制，其原因在于它们在类型表达式之间引入了 *equations*。系统 F 加上等式的扩展造就了“System Fc”，这是 GHC 如今所使用的系统。种类为  $*$  的类型表达式之间的等式 `s  $\tau$  t` 会诱导出强制转换，将值从 `s` 传送到 `t`。计算是通过从你在定义类型族时给出的规则中推导等式来完成的。

此外，你 *can* 可以为类型族指定一个高阶种类的返回类型，如下所示

类型族 `Hoo a` 类型实例 `Hoo Int = Maybe`  
类型实例 `Hoo Float = IO ...`

以便当 `t :: *` 时，`Hoo t :: * -> *`，但我们仍然不能让 `Hoo` 单独存在。  
我们有时用来绕过这一限制的技巧是 `newtype` 包装：

新类型 `Noo i = TheNoo {theNoo :: Foo i}`

这确实给了我们

`Noo :: * -> *`

但这意味着我们必须应用投影才能使计算发生，因此 `Noo Int` 和 `Int` 是可证明的不同类型，但是

`theNoo :: Noo 整数 -> 整数`

所以它有点笨拙，但我们可以在一定程度上弥补类型族与类型运算符在 F-omega 意义下不直接对应的事实。

## 7.21 如何推导具有非-\* 种类的幻影类型参数的 GADT 的公式

正如其他人所指出的，问题的关键在于Con3类型中的存在量化标签。当你试图定义

```
Con3 s == Con3 t = ???
```

没有理由认为 s 和 t 应该是具有相同标签的表达式。

但是也许你不在乎？你完全可以定义 *heterogeneous* 相等性测试，它很乐意在不考虑标签的情况下，结构性地比较表达式。

```
实例 Eq (Expr 标签) 其中 (==) = heq 其中 heq :: E
xpr a -> Expr b -> 布尔值 heq (Con1 i) (Con1 j) = i
== j heq (Con2 s) (Con2 t) = heq s t heq (Con3 s) (
Con3 t) = heq s t
```

如果你确实在意，那么你可能最好为 Con3 配备一个针对存在量化标签的运行时见证。实现这一点的方式是使用 *singleton* 构造。

```
数据 SingExprTag (tag :: ExprTag) 其中 SingTag1 :: Sin
gExprTag Tag1 SingTag2 :: SingExprTag Tag2
```

对SingExprTag标签中的值进行案例分析将准确地确定标签是什么。我们可以将这一额外的信息插入到Con3中，如下所示：

```
数据 Expr' (标签 :: ExprTag) 其中 Con1' :: 整数 -> Expr' 标签 Con2'
:: Expr' 标签 -> Expr' 标签 Con3' :: SingExprTag 标签 -> Expr' 标签 -
> Expr' 标签2
```

现在我们可以检查标签是否匹配。我们可以为标签单例编写一个异质相等，如下所示。。。

```
heqTagBoo :: SingExprTag a -> SingExprTag b -> 布尔 heqTagBoo SingT
ag1 SingTag1 = 真 heqTagBoo SingTag2 SingTag2 = 真 heqTagBoo __ =
假
```

...但是这样做是完全没有用的，因为它只给我们一个类型为 Bool 的值，完全不知道这个值意味着什么，也不清楚它的真实性可能赋予我们什么。知道 heqTagBoo a b = True 并不能告诉类型检查器关于 a 和 b 所见标签的任何有用信息。

布尔值的信息量有点少。

我们可以编写基本上相同的测试，但在正例情况下提供一些 *evidence*，表明这些标签是相等的。

```
数据 x :=: y 其中 Refl :: x
:=: x
```

```
singExprTagEq :: SingExprTag a -> SingExprTag b -> 也许 (a :=: b) singExprTagEq SingTag1 Si
ngTag1 = Just Refl singExprTagEq SingTag2 SingTag2 = Just Refl singExprTagEq __ = Nothing
```

现在我们终于进入正轨了！我们可以为 `Expr'` 实现一个 `Eq` 实例，它通过比较 `ExprTag` 来证明：当标签已经被证明匹配时，可以对两个 `Con3'` 子节点进行递归调用。

实例 `Eq (Expr' 标签)` 其中

```
Con1' i == Con1' j = i == j
Con2' s == Con2' t = s == t
Con3' a s == Con3' b t = case singExprTagEq a b of Just Refl -> s == t
                                Nothing -> False
```

总体情况是，被提升的类型需要它们关联的单例类型（至少在我们获得真正的  $\Pi$ -types 之前），并且我们需要能够产生证据的异质相等性测试来作用于这些单例族，这样当两个单例见证相同的类型层级值时，我们就可以比较它们并获得类型层级的知识。这样一来，只要你的 GADT 为任何存在量携带单例见证，你就可以进行同质的相等性测试，确保来自单例测试的正结果还能额外带来对其他测试中类型的统一。

## 7.22 哈斯克尔中的所有类型类都有类别理论的类比吗？

当以足够严谨的方式解读时，所有这些问题的答案都是“是”，但原因却是无关紧要的琐碎原因。

每个类别  $C$  限制为一个离散子类别  $|C|$ ，该子类别具有与  $C$  相同的对象，但只有恒等态射（因此没有有趣的结构）。至少，Haskell 类型上的操作可以无聊地解释为在离散类别  $|*|$  上的操作。最近的“角色”故事实际上（但并非以此方式表述）是试图承认态射是重要的，而不仅仅是对象。类型的“名义”角色意味着在  $|*|$  中工作，而不是在  $*$  中。

（注意，我不喜欢使用“Hask”作为“Haskell 类型和函数的范畴”名称：我担心将一个范畴标记为 *the Haskell* 范畴会产生不幸的副作用，使我们忽视了 Haskell 编程中丰富的 *other* 范畴结构。这是一个陷阱。）

作为一种不同的迂腐方式，我想指出，你可以随便编造任何类型类，作用于任何种类，完全没有有趣的结构（但仍然可以讨论那些微不足道的结构，若有人非得如此）。然而，你在库中找到的类型类通常是结构丰富的。作用于  $* \rightarrow *$  的类型类，通常设计上是 `Functor` 的子类，除了 `fmap` 之外，还要求存在某些自然变换。

对于问题 2。当然，在  $*$  上的一个类给出了  $*$  的一个子范畴。把对象从一个范畴中剔除并没有问题，因为范畴论中关于恒等和复合存在的要求，在给定对象的情况下只要求 *morphisms* 存在，但并不对哪些 *objects* 存在提出任何要求。它乏味地可能这一事实本身也是一个乏味的事实。然而，许多在  $*$  上的 Haskell 类型类所产生的范畴，要比仅仅作为  $*$  的子范畴得到的那些有趣得多。例如，`Monoid` 类给我们一个范畴，其中对象是 `Monoid` 的实例，而箭头是 *monoid homomorphisms*：不仅仅是从一个 `Monoid` 到另一个的任意旧函数 `f`，而是一个保持结构的函数：`f mempty = mempty`，并且 `f (mappend x y) = mappend (f x) (f y)`。

对于第 3 题来说，嗯，既然到处都潜藏着大量的范畴结构，那么在更高层次上肯定有大量的范畴结构可用（可能有，但不一定）。我特别喜欢索引集族之间的函子。

```
type (s :: k -> *) :-> (t :: k -> *) = forall x. s x -> t x
```

类 `FunctorIx (f :: (i -> *) -> (j -> *))` 其中 `mapIx :: (s -> t) -> (f s -> f t)`

当  $i$  和  $j$  重合时，询问这样的  $f$  何时是一个单子就变得合理。通常的范畴论定义就足够了，尽管我们已经把  $* \rightarrow *$  留在了身后。

信息是这样的：成为一个类型类这一事实本身并不会内在地诱导出 *interesting* 的范畴结构；有大量有趣的范畴结构，可以通过覆盖各种 *kind* 的类型类来有效地呈现。确实存在从  $*$ （集合与函数）到  $* \rightarrow *$ （函子与自然变换）的有趣函子。不要被对“Hask”的草率说法所蒙蔽，而忽视了 Haskell 中范畴结构的丰富性。

## 7.23 Haskell 类型解析在类型类中的应用（生成器，联合单子）

Reid 在他的评论中是对的。当你写

```
class Generator g where next :: St
ate g a 你真正想表达的是
```

类 Generator g 其中 next :: 对于所有 a. 状态 g a

以便从  $g$  中的给定状态，您的客户端可以生成他们希望的任何类型  $a$  的元素，而不是由  $g$  中的状态提供的任何类型。

有三种合理的方法来解决这个问题。我将按我偏好的顺序简要勾勒它们。方案 A 是认识到，任何事物的生成器在某种意义上都是它们的一个容器，因此应当作为类型构造子而不是类型来呈现。它当然应该是一个 Functor，并且很可能还是一个 Comonad。所以

```
类 Comonad f => 生成器 f 其中 move :: forall x. f x -> f x next
:: forall x. State (f x) x next = 状态 $ \ g -> ( 提取 g, 移动 g ) -
- 法则 -- 移动 . 重复 = 重复 . 移动
```

实例生成器 [] 其中移动 = 尾部

如果这些对你来说完全是天书，也许现在正是你按需学习一些新结构的机会！备选方案是忽略余单子结构并添加一个 *associated type*。

```
类 Generator g 其中类型 From g next
:: State g (From g)
```

```
实例生成器 [a] 其中类型 从 [a] = a 下一步 = 状态 $ \
(a : as) -> (a, as)
```

计划 C 是“功能依赖”版本，类似于 MonadSupply，如 Cirdec 所建议的。

```
类 Generator g a | g -> a 其中 next :: State g a
```

```
实例生成器 [a] a 其中下一个 = 状态 $ \ (a : as) -> (a,
as)
```

这些计划的共同点是，它们都承认了g和a之间的某种函数关系。如果没有这一点，什么也做不成。

## 7.24 证明类型级析取的幂等性

您请求的事情是不可能的，但类似的事情或许可以代替。之所以不可能，是因为证明需要在类型层面的布尔值上进行案例分析，但您没有数据能够使这种事件发生。解决方法是通过单例包含此类信息。

首先，让我指出你的 idemp 类型有点模糊。约束 `a b` 只是将同一件事重复命名。以下类型检查：

```
idempq :: p (或 b b) -> p b idempq = 未定义 idemp ::
a b => p (或 a b) -> p a idemp = idempq
```

如果你有一个约束 `a t`，其中 `t` 不包含 `a`，通常最好将 `t` 替换为 `a`。例外情况是在实例声明中：实例头中的 `a` 将匹配任何东西，因此即使该东西尚未明显变为 `t`，实例仍然会触发。但我有些离题了。

我声称 `idempq` 是无法定义的，因为我们对 `b` 没有有用的信息。唯一可用的数据存在于 `p-of-something` 中，而我们不知道 `p` 是什么。

我们需要根据 `b` 进行分类推理。请记住，对于一般的递归类型族，我们可以定义既不是 `True` 也不是 `False` 的类型级布尔值。如果我启用 `UndecidableInstances`，我可以定义

```
类型族 Loop (b :: Bool) :: Bool    类型实例 loop True
e = Loop False    类型实例 loop False = Loop True
```

因此，`Loop True` 不能简化为 `True` 或 `False`，更糟糕的是，没有办法证明这一点。

或者 (循环 真) (循环 真) 循环 真 -- 这不是真的

别无选择。我们需要运行时证据，证明我们的 `b` 是一个行为良好的布尔值，能够以某种方式计算得到一个值。因此让我们 *sing*

```
数据 Booly :: Bool -> * 其中 Truey :: Bool
ly True Falsey :: Booly False
```

如果我们知道 `Booly b`，我们就可以进行一次情形分析，从而告诉我们 `b` 是什么。随后每种情况都会顺利进行。下面是我会采用的做法：使用一个用 `PolyKinds` 定义的等式类型来把这些事实打包起来，而不是对 `p b` 的使用进行抽象。

```
数据 (:=:) a b 其中 Refl :: a :
:=: a
```

我们的关键事实现已被清楚地陈述并得到证明：

```
orIdem :: Booly b -> Or b b :=: b orIdem Truey
= Refl orIdem Falsey = Refl
```

并且我们可以通过严格的分类讨论来运用这一事实：

`idemp :: 布尔值 b -> p (或 b b) -> p` `idemp b p = 情况`  
`orIdem b` 为 `Refl -> p`

案例分析必须严格，以检查该证据并非某种荒谬的谎言，而是一个货真价实的 `Refl`，悄然只封装了修复类型所需的 `Or b b b` 的证明。

如果你不想显式地到处传递这些单例值，你可以像 `kosmikus` 建议的那样，把它们藏在一个字典里，在需要的时候再取出来。

Richard Eisenberg 和 Stephanie Weirich 有一个 Template Haskell 库，可以为你生成这些类型族和类。SHE 也可以构建它们，并让你编写

```
orIdem pi b :: Bool. Or b b :=: b orIdem {True}
= Refl orIdem {False} = Refl
```

其中 `pi b :: Bool` 展开为 `forall b :: Bool. Booly b ->`.

但这真是太麻烦了。这就是为什么我的团队正在致力于为 Haskell 添加一个实际的 `pi`，它是一个非参数量词（不同于 `forall` 和 `->`），可以通过现在 Haskell 的类型语言和项语言之间的非平凡交集的东西来实例化。这个 `pi` 还可以有一个“隐式”变体，其中参数默认保持隐藏。两者分别对应于使用单例族和类，但不需要定义三次数据类型来获得额外的工具。

值得一提的是，在全称类型理论中，*not* 需要在运行时传递布尔值 `b` 的额外副本。问题是，`b` 仅用于证明数据可以从 `p`（或 `b b`）传输到 `p b`，并不一定是为了传输数据本身。我们在运行时不会在绑定器下进行计算，因此无法编造出不诚实的方程证明，因此我们可以删除证明组件和传递它的 `b` 的副本。正如 Randy Pollack 所说，  
*the best thing about working in a strongly normalizing calculus is not having to normalize things.*

## 7.25 递归定义的实例和约束

`pure` 的定义确实处在问题的核心。它的类型应该是什么，如何做到完全量化并加以限定？

纯净 :: 对于所有  $(n :: \text{Nat}) (x :: *)$ .  $x \rightarrow \text{Vector } n$   $x$  -- (X)

行不通，因为在运行时没有可用的信息来确定 `pure` 应该产生 `VNil` 还是 `VCons`。相应地，在目前情况下，你不能就这样拥有

Applicative (Vector n) 的实例 -- (X)

你能做什么？好吧，在与 Strathclyde Haskell Enhancement 合作时，在 `Vec.lhs` 示例文件中，我定义了 `pure` 的前驱。

```
vec :: forall x. pi (n :: Nat). x -> Vector {n} x
vec {Zero} x = VNil
vec {Succ n} x = VCons x (vec n x)
```

使用一个 `pi` 类型，要求在运行时传递 `n` 的一个副本。这个 `pi (n :: Nat)` 会被去糖化为

对所有 `n`. `Natty n ->`

`Natty` 在现实生活中有一个更为平凡的名字，它是由以下给出的单例 GADT：

数据 `Natty n` 其中 `Zero :: Natty n` `Succ :: Natty n -> Natty n`

并且, `vec` 的方程中的花括号只是把 `Nat` 的构造子翻译成 `Natty` 的构造子。随后我定义了下面这个邪恶的实例 (关闭默认的 `Functor` 实例)。

```
实例 {n :: 自然数} => 应用式 (Vec {n}) 其中 隐藏 实例 函子 纯 = vec {n :: 自然数}
其中 (<*>) = vapp 其中 vapp :: Vec {m} (s -> t) -> Vec {m} s -> Vec {m} t
vapp VNil VN = VNil
vapp (VCons f fs) (VCons s ss) = VCons (f s) (vapp fs ss)
```

仍然需要进一步的技术。约束 `{n :: Nat}` 被简化为某种需要存在 `Natty n` 见证的形式, 通过作用域类型变量的力量, 相同的 `{n :: Nat}` 明确地传唤该见证。详细来说, 这是

```
class HasNatty n where natty :: Natty n
instance HasNatty Zero where natty = Zero
instance HasNatty n => HasNatty (Succ n) where natty = Succ natty
```

我们将约束 `{n :: Nat}` 替换为 `HasNatty n`, 并将相应的项替换为 `(natty :: Natty n)`。系统地进行这种构造相当于用类型类 `Prolog` 编写 `Haskell` 类型检查器的一个片段, 这不是我所认为的乐趣, 因此我使用计算机。

请注意, `Traversable` 实例 (请原谅我使用成语式的括号, 以及我默认为存在的 `Functor` 和 `Foldable` 实例) 并不需要这种花招

```
实例 Traversable (Vector n) 其中 traverse f VNil = (| VNil |)
traverse f (VCons x xs) = (| VCons (f x) (traverse f xs) |)
```

这就是你需要的所有结构, 以便在没有进一步显式递归的情况下进行矩阵乘法。

**TL;DR** 使用单例构造和其相关的类型类将所有递归定义的实例简化为类型级数据的运行时见证的存在, 从中可以通过显式递归进行计算。

设计含义是什么?

`GHC 7.4` 具有类型提升技术, 但 `SHE` 仍然提供单例构造 `pi` 类型。关于提升数据类型, 一个显然重要的事情是它们是 *closed*, 但这一点还没有完全显现出来: 单例见证的构造性是这种封闭性的表现。不知为何, 如果你有 `forall (n :: Nat)`。那么总是合理地要求一个单例, 但这样做会对生成的代码产生影响: 无论是像我的 `pi` 构造那样显式, 还是像 `{n :: Nat}` 中的字典那样隐式, 都会带来额外的运行时信息, 并相应地导致一个较弱的自由定理。

未来版本的 `GHC` 中一个开放的设计问题是如何管理类型级数据的运行时见证的存在与不存在之间的区别。一方面, 我们在约束中需要它们。另一方面, 我们需要对它们进行模式匹配。例如, `pi (n :: Nat)` 是否应该意味着显式的

对于所有  $(n :: \text{Nat})$ 。  $\text{Natty } n \rightarrow$

或隐式

对于所有  $(n :: \text{Nat})$ 。  $\{n :: \text{Nat}\} \Rightarrow$

? 当然, 像 Agda 和 Coq 这样的语言同时具备这两种形式, 所以也许 Haskell 也应该效仿。显然还有改进的空间, 我们正在努力!

## 7.26 如何获取依赖类型区间的长度?

这是我给你的程序版本。我正在使用

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures, TypeFamilies #-}
```

我有 Nat 及其单例

```
数据 Nat = Z | S Nat
```

```
数据 SNat :: Nat -> * 其中 ZZ :: SNat Z
```

```
SS :: SNat n -> SNat (S n)
```

您的区间类型对我来说更像是“小于或等于”的“后缀”定义: “后缀”是因为如果您将数字升级为列表, 并为每个 S 标记一个元素, 那么您就会得到列表后缀的定义。

```
数据 Le :: Nat -> Nat -> * 其中 Len :: SNat n ->
  > Le n n Les :: Le m n -> Le m (S n)
```

这是加法。

```
类型族 Plus (x :: Nat) (y :: Nat) :: Nat    类型实例 Plus Z y = y    类
型实例 Plus (S x) y = S (Plus x y)
```

现在, 你的谜题是对某个 Le 值中的 Le 构造子进行计数, 提取其索引差异对应的单例。不同于 *assuming* 我们处理某个  $\text{Le } n (\text{Plus } m \ n)$  并试图计算一个  $\text{SNat } m$ , 我将编写一个函数, 用来计算 *arbitrary*  $\text{Le } m$  的  $o$ -索引之间的差异, 以及 *establishes* 与 Plus 的关联。

这是 Le 的加性定义, 其中补充了单元元素集。

```
数据 AddOn :: Nat -> Nat -> * 其中 AddOn :: SNat n -> SNat m -> Add
  On n (Plus m n)
```

我们可以轻松地证明 Le 蕴含 AddOn。对某些  $\text{AddOn } n \ o$  进行模式匹配, 揭示  $o$  为  $\text{Plus } m \ n$ , 其中  $m$  为某个值, 并为我们提供了所需的单例。

```
leAddOn :: Le m o -> AddOn m o leAddOn (Len n) = AddOn n ZZ leAddOn (Les p) = case l
eAddOn p of AddOn n m -> AddOn n (SS m)
```

更一般地说，我建议表述依赖类型编程问题时，尽量减少在你计划进行模式匹配的类型索引中出现已定义函数。这样可以避免复杂的统一。（Epigram 曾用绿色来标示这类函数，因此有了“别碰绿色黏液！”的忠告。）事实证明，`Len o`（这正是 `leAddOn` 的要点）并不比 `Len (Plus m n)` 提供的信息更少，但它在进行匹配时要容易得多。

更一般地说，建立一个能够刻画问题逻辑的依赖数据类型，却在使用上极其糟糕，这是相当常见的经历。这并不意味着所有能够刻画正确逻辑的数据类型在使用上都会非常糟糕，而只是说明你需要更加深入地思考你所做定义的易用性。把这些定义整理得干净利落，并不是很多人在日常函数式编程学习过程中就能掌握的技能，因此要预期将面对一条新的学习曲线。

## 7.27 如何使卡塔变换与参数化/索引类型一起工作？

我在2009年写了一篇关于这个主题的演讲，叫做“切片”。它显然指向我在斯特拉斯克莱德大学的同事Johann和Ghani关于GADT的初始代数语义的工作。我使用了SHE提供的符号来编写数据索引类型，但令人欣慰的是，这已经被“提升”理论所取代。

演讲的关键点是，根据我的评论，要系统地使用一个精确的索引，但要利用其类型可以变化这一事实。

所以实际上，我们有（使用我目前偏好的“戈希尼和尤德佐”名字）

类型  $s \rightarrow t = \text{对所有 } i. s\ i \rightarrow t\ i$

类 `FunctorIx f` 其中 `mapIx :: (s -> t) -> (f s -> f t)`

现在你可以证明 `FunctorIx` 在固定点下是 *closed*。关键是将两个索引集结合成一个，提供选择索引的功能。

数据案例  $(f :: i \rightarrow *) (g :: j \rightarrow *) (b :: \text{Either } i\ j) :: *$  其中  $L :: f\ i \rightarrow \text{案例 } f\ g\ (\text{左 } i)\ R :: g\ j \rightarrow \text{案例 } f\ g\ (\text{右 } j)$

$(\langle? \rangle) :: (f \rightarrow f') \rightarrow (g \rightarrow g') \rightarrow \text{情况 } f\ g \rightarrow \text{情况 } f'\ g' \quad (f \langle? \rangle g) (L\ x) = L (f\ x)$   
 $(f \langle? \rangle g) (R\ x) = R (g\ x)$

现在我们可以获取那些“包含元素”代表“有效载荷”或“递归子结构”的函子的定点。

数据  $\text{MuIx } (f :: (\text{Either } i\ j \rightarrow *) \rightarrow j \rightarrow *) :: (i \rightarrow *) \rightarrow j \rightarrow *$  其中  $\text{InIx} :: f (\text{Case } x (\text{MuIx } f\ x))\ j \rightarrow \text{MuIx } f\ x\ j$

因此，我们可以对“payload”进行 `mapIx`。...

实例 `FunctorIx f => FunctorIx (MuIx f)` 其中 `mapIx f (InIx xs) = InIx (mapIx (f <?> mapIx f) xs)`

... 或者在“递归子结构”上编写一个双向同态。...

`foldIx :: FunctorIx f => (f (Case x t) -> t) -> MuIx f x -> t` `foldIx f (InIx xs) = f (mapIx (id <?> foldIx f) xs)`

... 或者两者同时。

```
mapFoldIx :: FunctorIx f => (x :-> y) -> (f (Case y t) :-> t) -> MuIx f x :-> t
mapFoldIx e f (InIx xs) = f (mapIx (e <?> mapFoldIx e f) xs)
```

FunctorIx 的乐趣在于它具有如此出色的闭包性质，这要归功于能够变化索引的种类。MuIx 允许引入有效载荷的概念，并且可以进行迭代。因此，相较于使用多个索引，采用结构化索引更具吸引力。

## 7.28 在签名中约束构造函数

你~~can~~用 GADTs 做这种事情。我无意评判结果是否会变成一个兔子洞，但至少让我展示一下配方。我使用了新的 PolyKinds 扩展，但用更少的也能应付。

首先，确定你需要哪些类型的内容，并为这些类型定义一种数据类型。

数据 排序 = 基础 | 复合

接下来，按它所属的 sort 对数据进行索引并加以定义。这就像在构建一种小型的类型化语言。

数据 武器部件 :: 排序 -> \* 其中

WInt :: Int -> 武器部件 基础 WHash :: Map.Map String Int -> 武器部件 基础 WNull :: 武器部件 基础

WTrans :: (Some WeaponPart -> Some WeaponPart) -> 武器部件 复合

你可以重新表示~~sort~~通过存在量化来表示“任何类型的数据”，as fo      It seems like the source text you wanted to transla

数据 Some p 其中

威特 :: p x -> 一些 p

请注意，x 并没有逃逸，但我们仍然可以检查 x “满足” p 的“证据”。请注意，Some 必须是一个数据类型，而不是一个新类型，因为 GHC 反对存在量新类型。

你现在可以自由地编写 Sort-泛型操作。如果你有泛型输入，你可以直接使用多态性，实际上是将 Some p -> ... 柯里化为 forall x. p x -> ....

实例 Show (WeaponPart x) 在这里 show (WInt x) = "WInt "

++ (show x) show (WHash x) = "WHash " ++ (show x)

show (WTrans \_) = "WTrans" show WNull = "WNull"

存在                  需要用于排序通用输出：这里我将它用于输入                  t 和输出。

```
冰冷 :: 某个 武器部件 -> 某个 武器部件
冰冷 (Wit (WInt x)) = Wit (WHash (Map.singleton "frost" x))
冰冷 (Wit (WHash x)) = Wit (WHash $ Map.insertWith (+) "frost" 5 x)
冰冷 (Wit (WTrans x)) = 冰冷 $ x (Wit (WInt 5))
冰冷 (Wit WNull) = 冰冷 $ Wit (WInt 5)
```

我不得不在各处偶尔加点机智的点缀，但程序还是同一个。与此同时，我们现在可以写下

```
ofTheAbyss :: WeaponPart 复合 -> 某个 WeaponPart ofTheAbyss (WTrans x) = x (Wit (WTrans x))
```

所以，使用嵌入式类型系统并不是非常糟糕。有时是有成本的：如果你希望你的嵌入式语言具有 *subsorting*，你可能会发现你需要额外的计算来仅仅改变一些数据类型的索引，这对数据本身没有任何影响。如果你不需要子排序，这种额外的规范通常是一个真正的朋友。

## 7.29 类似 Either 但用于 3 种情况的和类型的标准名称？

在近年的 Haskell 中，我会开启一点“全家桶”。

{-# LANGUAGE 多态种类, 数据种类, 广义代数数据类型, 种类签名, 类型运算符, 模式同义词 #-}

然后我会定义类型级别的列表成员资格

```
数据 (:>) :: [x] -> x -> * 其中 Ze :: (x' : xs) :>
x Su :: xs :> x -> (y' : xs) :> x
```

现在我已经得到了所有的有限和，而不需要列出一大堆OneOfN类型的定义：

```
数据求和 :: [*] -> * 其中 (-:) :: xs :> x -> x ->
求和 xs
```

但是，为了解决Tomas关于可读性的问题，我会利用模式同义词。实际上，这正是我多年来一直在强调模式同义词的原因。你可以有一个有趣的Maybe版本：

类型 MAYBE x = 总和 ' [(), x]

模式 无 :: 也许 x 模式 无 = Ze :- ()

模式 JUST :: x -> 可能 x 模式 JUST x =  
苏泽 :- x

而且你甚至可以使用 newtype 来构建递归和类型。

新类型 Tm x = Tm (和 ' [x, (Tm x, Tm x), Tm (也许 x)])

模式 VAR :: x -> Tm x 模式 VAR x = T  
m (Ze :- x)

模式 APP :: Tm x -> Tm x -> Tm x 模式 APP f s = Tm  
(Su Ze :- (f, s))

模式 LAM :: Tm (也许 x) -> Tm x 模式 LAM b = T  
m (Su (Su Ze) :- b)

新的类型包装器还允许你为以这种方式构建的类型声明实例。

当然，你也可以使用模式同义词来优雅地隐藏一个嵌套的 Either。

这种技术并非只适用于求和：它同样适用于乘积，而这基本上正是 de Vries 和 Löh 的 Generics-SOP 库中所发生的事情。

通过这种编码获得的巨大优势是，数据的描述本身就是（类型级别的）数据，使得你可以在不修改编译器的情况下，构造大量类似派生的功能。

在未来（如果由我说了算），所有数据类型都将是 *defined*，而不是 *declared*；其数据类型描述将由数据构成，这些数据同时指定数据的代数结构（从而允许计算通用操作）以及其外观（这样在处理某一特定类型时，你可以直观地看到自己在做什么）。

但未来已经有点到来了。

## 7.30 如何为以 GADT 形式定义的 AST 中的异构集合指定类型？

解决这个问题的一种方法是给值打上运行时类型代表的标签。我这里是在借鉴 Stephanie Weirich 的思路。让我们看一个小例子。首先，为一些类型给出表示。这通常通过一个 *singleton* 构造来完成。

```
数据 类型 :: * -> * 其中 Int :: 类型 Int Ch
ar :: 类型 Char List :: 类型 x -> 类型 [x]
```

因此，`Type Int` 包含一个值，我也把它称为 `Int`，因为它充当类型 `Int` 的运行时代表。如果你即使在单色事物中也能看出颜色，那么 `::` 左边的 `Int` 是红色的，而 `Type` 之后的 `Int` 是蓝色的。

现在我们可以进行存在打包，同时保持效用。

```
数据 Cell :: * 其中 (:) :: x -> 类型 x -> C
ell
```

`Cell` 是一个带有其类型的运行时代表的值。你可以通过读取其类型标记来恢复该值的效用。事实上，由于类型是一阶结构，我们可以以一种有用的方式检查它们是否相等。

```
数据 EQ :: k -> k -> * 其中 Refl :: EQ x
x
```

```
typeEQ :: 类型 x -> 类型 y -> 也许 (EQ x y) typeEQ Int Int = Just
Refl typeEQ Char Char = Just Refl typeEQ (List s) (List t) = 情况 t
typeEQ s t 为 Just Refl -> Just Refl Nothing -> Nothing typeEQ _ _
= Nothing
```

布尔等式在类型代表上没有用处：我们需要等式测试来构造可以统一的 *evidence*。通过生成证据的测试，我们可以写出

```
gimme :: 类型 x -> 单元 -> 可能 x gimme t (x :: s) =
情况 typeEQ s t 的 Just Refl -> Just x Nothing -> Nothi
ng
```

当然，写类型标签很麻烦。但是，为什么要养狗还自己叫呢？

```
class TypeMe x where myTy
pe :: Type x
```

```
实例 TypeMe Int 其中 myType = Int
```

```
实例 TypeMe Char, 其中 myType =
Char
```

### 7.31. TYPE-THREADED HETEROGENOUS LISTS AND DEFAULTING(?) WITH TYPE FAMILIES?189

实例 `TypeMe x => TypeMe [x]` 其中 `myType = 列表`  
`myType`

```
cell :: TypeMe x => x -> Cell
cell x = x :: myType
```

现在我们可以做一些事情，比如

```
myCells :: [Cell]
myCells = [cell (length "foo"), cell "foo"]
```

然后得到

```
> 给我 Int (head myCells) Just 3
```

当然，如果我们不必进行单例构造，并且可以在运行时选择保留的类型上进行模式匹配，那一切会变得更加简洁。我预计当神话般的 `pi` 量词变得不再那么神话时，我们会达到这个目标。

## 7.31 类型线程异构列表和类型族的默认值(?)

是否有某种特定原因导致 *Kleene star* GADT 无法完成这项工作？

数据 `Star r a b` 其中 `Nil :: Star r a a` `Cons :: r a b -> Star r b c -> Star r a c`

```
compose :: Star (->) a b -> a -> b
compose Nil = id
compose (Cons f fs) = compose fs . f
```

但如果你需要一种类型类的方法，我不会干涉。

## 7.32 构造函数，通过 `DataKinds` 提升到 $* \rightarrow A$

目前恐怕不行。我还没有发现明显的解决方法。

此票据记录了数据种类声明的前景，出生种类，而非被强加给它们的友善数据类型。对于此类事物的构造者来说，将类型打包成你所提议的样子是完全合理的。我们还没到达那一步，但看起来并不太成问题。

我的眼睛盯着更大的奖项。我希望 $*$ 成为完全合理的运行时值类型，这样您想要的类型可以通过提升的方式存在，正如我们今天所拥有的那样。将其与提到的 $\pi$ 类型的概念（在类型和值共享的语言部分上进行的非参数化抽象）结合，我们可能会得到一种比现在使用 `Data.Typeable` 更直接的方式来进行临时类型抽象。通常的 `forall` 将保持为参数化。

## 7.33 应如何理解“lemma”函数的一般类型？

我们收到了一些出色的答案，但作为施害者，我想提供一些评论。

是的，这些引理有多种等效的表示方式。我使用的表示方式是其中之一，选择主要是出于实际考虑。现在（在一个更新的代码库中），我甚至会定义

```
-- Holds :: 约束 -> * 类型 Holds c = 对于所有 t . (c => t)
-> t
```

这是一个*eliminator type*的例子：它抽象了它所提供的内容（消除的*motive*），并且要求你构建零个或多个*methods*（，这里）是在更具体的情况下实现动机。阅读它的方式是*backwards*。它说

如果你有一个问题（为任何动机类型 *t* 构造居元），而且没有其他人能帮你，也许你可以通过在你的方法中假设约束 *c* 来取得进展。

鉴于约束的语言允许合取（即元组化），我们获得了写出如下形式引理的手段

```
lemma :: forall x1 .. xn. (p1[x1 .. xn],... pm[x1 .. xn])      -- premises
                        => t1[x1 .. xn] -> .. t1[x1 .. xn]      -- targets
                        -> Holds (c1[x1 .. xn],... ck[x1 .. xn]) -- conclusions
```

并且可能某个约束，前提 *p* 或结论 *c*，具有方程式的形式

```
l[x1 .. xn] ~ r[x1 .. cn]
```

现在，为了部署这样的词条，考虑填补一个空洞的问题

`_ :: 问题`

通过消去引理来优化此问题，指定*targets*。*motive*来自当前问题。*Holds*情况下的*method*（奇异性仍然未解决。

引理 目标1 .. 目标l \$ \_

并且方法孔的类型不会发生变化

`_ :: 问题`

但是GHC会知道更多的东西，因此更有可能相信你的解决方案。

有时，在选择作为（约束）前提和作为（数据）目标之间，需要做出选择。我倾向于选择这些，以避免歧义（西蒙喜欢猜测 *x1 .. xn*，但有时需要提示），并促进*proof by induction*，这对于目标（通常是类型级数据的单例）来说，比对于前提要容易得多。

在部署方面，对于方程，你当然可以切换到数据类型的呈现方式，并展开分类讨论

案例数据Lemma 目标1 .. 目标l 的Refl -> 方法

而实际上，如果你为自己配备了Dict存在论

数据 Dict (c :: 约束) :: \* 其中 Dict :: c => Dict c

你可以一次做很多事情

案例 multiLemma blah blah blah of (Refl, Dict, Dict, Refl) -> 方法

### 7.33. HOW SHOULD THE GENERAL TYPE OF A “LEMMA” FUNCTION BE UNDERSTOOD?191

但消去子形式更加紧凑且更具可读性 *when there is at most one method*。事实上，我们可以将多个引理串联起来，而无需不断向右滑动。

引理1 .. \$ ... 引理j  
.. \$ 方法

如果你有一个包含两个或更多情况的消除器，我认为通常最好将其包装为GADT，以便使用位置能够通过构造函数标签帮助标记每种情况。

无论如何，是的，关键在于选择一种最简洁的事实呈现方式，使我们能够扩展GHC的约束求解机制，以便更多的内容能够通过类型检查。如果你和Simon发生争执，通常一个好的策略是向隔壁的Dimitrios解释自己。



## 第八章

# 类型理论

### 8.1 直觉主义类型理论的组合逻辑等价是什么？

所以我再多想了一下，并取得了一些进展。这是首次尝试以组合风格编码 Martin-L öf 的简单（但不一致的）集合：集合系统。这不是一个好的完成方式，但它是最容易开始的地方。这个类型理论的语法只是带有类型注释、Pi类型和一个宇宙集合的  $\lambda$ -演算。

#### 目标类型理论

为了完整性，我将呈现规则。上下文有效性只是说你可以通过连接新的变量来从空集构建上下文，这些变量存在于集合中。

$$\frac{}{. \vdash \text{valid}} \quad \frac{G \vdash \text{valid} \quad G \vdash S : \text{Set}}{G, x:S \vdash \text{valid}} \quad x \text{ fresh for } G$$

现在我们可以说明如何在任何给定的上下文中为术语合成类型，以及如何根据术语所包含的计算行为改变某个事物的类型。

$$\frac{G \vdash \text{valid}}{G \vdash \text{Set} : \text{Set}} \quad \frac{G \vdash S : \text{Set} \quad G \vdash T : \text{Pi } S \setminus x:S \rightarrow \text{Set}}{G \vdash \text{Pi } S T : \text{Set}}$$
$$\frac{G \vdash S : \text{Set} \quad G, x:S \vdash t : T x}{G \vdash \setminus x:S \rightarrow t : \text{Pi } S T} \quad \frac{G \vdash f : \text{Pi } S T \quad G \vdash s : S}{G \vdash f s : T s}$$
$$\frac{G \vdash \text{valid} \quad x:S \text{ in } G}{G \vdash x : S} \quad \frac{G \vdash s : S \quad G \vdash T : \text{Set}}{G \vdash s : T} \quad S = \{\text{beta}\} T$$

与原始版本略有不同，我将 `lambda` 作为唯一的绑定算子，因此 `Pi` 的第二个参数应当是一个函数，用来计算返回类型如何依赖于输入。按照约定（例如在 `Agda` 中，但遗憾的是在 `Haskell` 中并非如此），`lambda` 的作用域会尽可能向右延伸，所以当它们是高阶算子的最后一个参数时，通常可以不加括号地书写抽象：你可以看到我在 `Pi` 上就是这样做的。你的 `Agda` 类型 `(x : S) -> T` 变成了 `Pi S \ x:S -> T`。

(*Digression*对lambda的类型注解是必要的，如果你希望能够*synthesize*抽象的类型。如果你切换到将类型*checking*作为操作模式，你仍然需要注解来检查像 $(\lambda x \rightarrow t) s$ 这样的beta-简化项，因为你无法从整体的类型推测出各部分的类型。我建议现代设计者检查类型，并从语法中排除beta-简化项。

(*Digression*。该系统是不一致的，因为Set:Set允许编码各种“说谎悖论”。当Martin-Löf提出这个理论时，Girard将其在他自己不一致的系统U中进行了编码。Hurkens随后提出的悖论是我们所知道的最巧妙的有毒构造。)

### 组合子语法与标准化

总之，我们有两个额外的符号，Pi 和 Set，因此也许可以用 S、K 以及两个额外符号来实现一种组合式翻译：我选择用 U 表示宇宙，用 P 表示积。

现在我们可以定义无类型的组合语法（带有自由变量）：

数据 SKUP = S | K | U | P 派生 (Show, 式)

数据 Unty a = C SKUP | Unty a ::

Unty a | V a 派生 (Functor, Eq) inf

ixl 4 ::

请注意，我已经在此语法中包括了表示自由变量的方式，这些变量由类型 a 表示。除了我个人的反思（任何值得拥有名称的语法都是一个自由单子，其中 return 嵌入变量并且 >>= 执行替换），它将有助于表示在将带绑定的项转换为组合形式的过程中间阶段。

这里是归一化：

范数 :: Unty a -> Unty a 范数 (f :: a) =

范数 f \$. a 范数 c = c

(\$.) :: Unty a -> Unty a -> Unty a -- 要求第一个参数为正规形 C S :: f :: a \$. g = f \$. g \$. (a :: g) -- S f a g = f g (a g) 共享环境 C K :: a \$. g = a -- K a g = a 丢弃环境 n \$. g = n :: norm g -- 保证输出为正规形 infixl 4 \$.

（留给读者的一个练习是为恰好是规范形的项定义一个类型，并细化这些操作的类型。）

### 表示类型论

我们现在可以为我们的类型论定义一种语法。

data Tm a = Var a | Lam (Tm a) (Tm (Su a)) -- Lam 是唯一发生绑定的地方 | Tm a :: Tm a | Pi (Tm a) (Tm a) -- Pi 的第二个参数是一个计算 Set 的函数 | Set deriving (Show, Functor)

## 8.1. WHAT IS THE COMBINATORY LOGIC EQUIVALENT OF INTUITIONISTIC TYPE THEORY?195

infixl 4 :\$

数据 Ze 魔法 :: Ze -> 一个魔法 x = x ‘seq ‘  
错误 "悲剧!"

数据 Su a = Ze | Su a deriving (Show, Functor, Eq)

我使用 Bellegarde 和 Hook 风格的 de Bruijn 索引表示法（由 Bird 和 Paterson 推广）。类型 Su a 比 a 多一个元素，我们将其用作绑定符下的自由变量类型，其中 Ze 是新绑定的变量，Su x 是旧自由变量 x 的移位表示。

将术语翻译为组合子

至此，我们获得了基于 *bracket abstraction* 的常规翻译。

```
tm :: Tm a -> Unty a
tm (Var a) = V a
tm (Lam _ b) = bra (tm b)
tm (f :$ a) = tm f :. tm a
tm (Pi a b) = C
P :. tm a :. tm b tm Set = C U
```

bra :: Unty (Su a) -> Unty a -- 绑定一个变量，构造一个函数  
bra (V Ze) = C S :. C K :. C K -- 变量自身产生恒等  
bra (V (Su x)) = C K :. V x -- 自由变量变为常量  
bra (C c) = C K :. C c -- 组合子变为常量  
bra (f :. a) = C S :. bra f :. bra a -- S 正是被提升的应用

输入组合子

翻译展示了我们使用这些组合子的方式，这为它们的类型应该是什么提供了相当多的线索。U 和 P 只是集合构造子，因此，写下未翻译的类型并允许对 Pi 使用“Agda 记法”，我们应该有

U : 集合 P : (A : 集合) -> (B : (a : A) -> 集合) -> 集合

K 组合子用于将某种类型 A 的值提升为一个关于另一种类型 G 的常量函数。

G : 集合 A : 集合  
Sure, please provide the source text you'd like me to translate.  
K : (a : A) -> (g : G) -> A

S 组合子用于将应用提升到一个类型上，所有部分都可能依赖于该类型。

G : 集合 A : (g : G) -> 集合 B : (g : G) -> (a : A g) -> 集合

Sure! Please provide the source text you would like me to translate to Chinese (zh).  
S : (f : (g : G) -> (a : A g) -> B g a) -> (a : (g : G) -> A g) -> (g : G) -> B g (a g)

如果你看看S的类型，你会发现它正好阐明了类型理论中的*contextualised*应用规则，因此这就是它能够反映应用构造的原因。这就是它的工作！

我们然后只对封闭的事物进行应用。

$f : \Pi A B a : A$

请提供需要翻译的源文本。

$f a : B a$

但是有一个问题。我写的是普通类型理论中的组合子类型，而不是组合类型理论。幸运的是，我有一台可以进行翻译的机器。

组合子类型系统

-----  $U : U$  -----  $P : PU(S(S(KP)(S(S(KP)(SKK))(S(KK)(KU))))(S(KK)(KU))) G : U A : U$

Please provide the source text you would like translated to Chinese.

$K : P[A](S(S(KP)(K[G]))(S(KK)(K[A])))$

$G : U A : P[G](KU) B : P[G](S(S(KP)(S(K[A])(SKK)))(S(KK)(KU)))$

Understood. Please provide the source text you would like to have translated into Chinese.

$S : P(P[G](S(S(KP)(S(K[A])(SKK)))(S(S(KS)(S(S(KS)(S(KK)(K[B]))(S(KK)(SKK))))(S(S(KS)(KK))(KK))))(S(S(KP)(S(S(KP)(K[G]))(S(S(KS)(S(KK)(K[A]))(S(S(KS)(KK))(KK))))(S(S(KS)(S(S(KS)(S(KK)(KP)))(S(KK)(K[G]))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(KK)(KK)))(S(KK)(K[B]))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(KK)(KK))))(S(KK)(KK))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(KK)(KK)))(S(S(KS)(KK))(KK)))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(KK)(KK))))(S(KK)(KK))))))$

$M : A B : U$

-----  $A = \{\text{norm}\} B$

$M : B$

所以，这就是它了，以其全部难以读懂的“辉煌”：Set:Set 的一种组合子式表述！

仍然存在一些问题。系统的语法没有提供任何方式，仅通过项来推测S的G、A和B参数，K也是如此。相应地，我们可以通过*typing derivations*算法进行验证，但不能像原系统那样对组合子项进行类型检查。可能可行的方法是要求类型检查器的输入对S和K的使用进行类型注解，实际上记录推导过程。但那是另一个复杂的问题……

这是一个很好的停止点，如果你足够敏锐去开始的话。剩下的都是“幕后”的内容。

生成组合子的类型

我使用相关类型论项的括号抽象翻译生成了那些组合子类型。为了展示我是如何做到的，并且让这篇文章不至于完全没有意义，让我来提供

## 8.1. WHAT IS THE COMBINATORY LOGIC EQUIVALENT OF INTUITIONISTIC TYPE THEORY?197

我的设备。

我可以按如下方式编写组合子的类型，完全抽象化它们的参数。我使用我方便的 `pil` 函数，它结合了 `Pi` 和 `lambda`，避免了重复域类型，并且相当有用地允许我使用 Haskell 的函数空间来绑定变量。也许你几乎可以读懂下面的内容！

```
pTy :: Tm a pTy = fmap magic $ pil Set $ \_A -> pil (pil _A $ \_ -> Set) $ \_B -> Set
```

```
kTy :: Tm a kTy = fmap magic $ pil Set $ \_G -> pil Set $ \_A -> pil _A $ \a -> pil _G $ \g -> _A
```

```
sTy :: Tm a
sTy = fmap magic $ pil Set $ \_G -> pil (pil _G $ \g -> Set) $ \_A -> pil (pil _G $ \g -> pil (_A :$ g
) $ \_ -> Set) $ \_B -> pil (pil _G $ \g -> pil (_A :$ g) $ \a -> _B :$ g :$ a) $ \f -> pil (pil _G $ \g -
> _A :$ g) $ \a -> pil _G $ \g -> _B :$ g :$ (a :$ g)
```

在这些定义完成后，我提取了相关的 *open* 子术语，并对它们进行了翻译。

一个 de Bruijn 编码工具包

以下是构建 `pil` 的方法。首先，我定义了一类有限集合，用于表示变量。每个这样的集合都有一个保持构造子的嵌入，嵌入到其上方的集合中，并额外增加一个新的顶元素，而且你可以区分它们：`embd` 函数会告诉你一个值是否位于 `emb` 的像中。

```
类 Fin x, 其中 top :: Su x emb :: x -
> Su x embd :: Su x -> Maybe x
```

我们当然可以为 `Ze` 和 `Suc` 实例化 `Fin`。

实例 `Fin Ze` 其中

顶部 = `Ze` -- `Ze` 是唯一的，因此是最高的 `emb` = 魔法嵌入 `_` = 无 -- 没有什么可以嵌入的

实例 `Fin x => Fin (Su x)` 其中

`top = Su top` -- 最高的要高一阶 `emb Ze = Ze` -- `emb` 保留 `Ze` `emb (Su x) = Su (emb x)` --  
且 `Su embd Ze Just Ze = Just Ze` -- `Ze` 确实是嵌入的 `embd (Su x) = fmap Su (embd x)` --  
否则，静观其变

现在我可以使使用 *weakening* 操作定义小于等于。

类 `(Fin x, Fin y) => Le x y` 在哪里

```
wk :: x -> y
```

wk 函数应当将  $x$  的元素嵌入为  $y$  的 *largest* 元素, 这样  $y$  中的额外东西更小, 因此用 de Bruijn 索引的术语来说, 绑定得更局部。

实例  $\text{Fin } y \Rightarrow \text{Le } \text{Ze } y$  其中  $\text{wk} = \text{magic}$  -- 无需嵌入 实例  $\text{Le } x \ y \Rightarrow \text{Le } (\text{Su } x) (\text{Su } y)$  其中  $\text{wk } x =$   
 案例  $\text{embd } x \text{ of Nothing} \rightarrow \text{top} \text{ -- top 映射到 } \text{top Just } y \rightarrow \text{emb } (\text{wk } y) \text{ -- 嵌入被削弱并且嵌入}$

一旦你把这个解决了, 稍微一点  $n$  阶诡计就能搞定其余的。

```
lam :: forall x. Tm x -> ( (forall y. Le (Su x) y => Tm y) -> Tm (Su x)) -> Tm x
lam s f = Lam s (f (Var (wk (Ze :: Su x))))
pil :: forall x. Tm x -> ( (forall y. Le (Su x) y => Tm y) -> Tm (Su x)) -> Tm x
pil s f = Pi s (lam s f)
```

这个高阶函数不仅仅给你一个表示该变量的项, 它还给你一个 *overloaded* 的东西, 在变量可见的任何作用域中, 它都会成为该变量的正确表示。也就是说, 我费力去区分不同的作用域 *by type* 这一事实, 为 Haskell 的类型检查器提供了足够的信息, 以计算在翻译为 de Bruijn 表示时所需的位移。何必养条狗却自己叫?

## 8.2 Hask 或 Agda 有等化子吗?

简而言之, 所提出的候选项并不完全是一个均衡器, 但其无关的对应项却是。

候选的 Agda 均衡器看起来不错。那么我们就试试看。我们需要一些基本工具。这是我拒绝者的 ASCII 依赖对类型和同质的强加性相等。

```
记录 Sg (S : 集合) (T : S -> 集合) : 集合 其中 构造器 _,_ 字段 fst
: S snd : T fst 打开 Sg 数据 _==_ {X : 集合} (x : X) : X -> 集合
其中 refl : x == x
```

这是您为两个函数的均衡器候选者

```
Q : {S T : 集合} (f g : S -> T) -> 集合
Q {S} {T} f g = Sg S \ s -> f s == g s
```

通过  $\text{fst}$  投影将  $Q \ f \ g$  发送到  $S$ 。

它所说的是:  $Q \ f \ g$  的一个元素是源类型的一个元素  $s$ , 以及一个证明  $f \ s == g \ s$  的证据。但这只是一个等化子吗? 让我们试着使它成为等化子。

要说明什么是均衡器, 我应该先定义函数组合。

```
_o_ : {R S T : 集合} -> (S -> T) -> (R -> S) -> R -> T
(f o g) x = f (g x)
```

待修复: <http://i.stack.imgur.com/odrtv.jpg>

图 8.1: 均衡器图示

所以现在我需要证明: 任何  $h : R \rightarrow S$ , 只要它使  $f \circ h$  与  $g \circ h$  相同, 就必须通过候选的  $\text{fst} : Q \text{ f g } \rightarrow S$  分解。我需要给出另一个组成部分  $u : R \rightarrow Q \text{ f g }$ , 以及证明  $h$  的确可以分解为  $\text{fst} \circ u$ 。图示如下:  $(Q \text{ f g}, \text{fst})$  是一个等化子, 如果在没有  $u$  的情况下该图表交换, 那么就存在唯一一种方式加入  $u$ , 并且图表仍然交换。

这里是中介  $u$  的存在。

```
mediator : {R S T : Set} (f g : S -> T) (h : R -> S) ->
  (q : (f o h) == (g o h)) ->
  Sg (R -> Q f g) \ u -> h == (fst o u)
```

显然, 我应该选择  $h$  所选择的  $S$  中的同一个元素。

中介  $\text{f g h q} = (\lambda r \rightarrow (h r, ?0)), ?1$

这使我留下了两个证明义务

```
?0 : f (h r) == g (h r) ?1 : h == \
r -> h r
```

现在,  $?1$  可以直接作为  $\text{refl}$ , 因为 Agda 的定义等式具有函数的 eta 法则。对于  $?0$ , 我们有  $q$  的帮助。相等的函数遵循应用规律。

```
funq : {S T : 集合} {f g : S -> T} -> f == g -> (s : S) -> f s == g s
funq refl s = refl
```

因此我们可以取  $?0 = \text{funq } q \text{ r}$ 。

但我们不要过早庆祝, 因为仅仅存在一个中介态射还不够。我们还要求它的唯一性。而问题很可能在这里出岔子, 因为  $==$  是 *intensional*, 所以唯一性意味着始终只有一种方式来 *implement* 这个中介映射。但接着, 我们的假设也是内涵性的。...

这是我们的证明义务。我们必须证明任何其他中介态射都等于由中介器选定的那个。

```
mediatorUnique :
  {R S T : 集合} (f g : S -> T) (h : R -> S) -> (qh : (f o h)
    ) == (g o h) -> (m : R -> Q f g) -> qm : h == (fst o m)
    -> m == fst (中介 f g h qh)
```

我们可以立即通过  $qm$  代入并得到

```
mediatorUnique f g .(fst o m) qh m refl = ?
```

```
? : m == (\ r -> (fst (m r) , funq qh r))
```

这看起来不错, 因为 Agda 对记录有  $\eta$  定律, 所以我们知道

```
m == (\ r -> (fst (m r) , snd (m r)))
```

但当我们尝试使  $? = \text{refl}$  时, 我们收到投诉

`snd (m _) != funq qh _` 的类型为 `f (fst (m _)) == g (fst (m _))`

这很烦人，因为在标准配置下，同一性证明是唯一的。现在，你可以通过假定外延性，并利用关于相等性的其他一些事实来摆脱这一困境。

假设 `ext : {S T : 集合} {f g : S -> T} -> ((s : S) -> f s == g s) -> f == g`

`sndq : {S : Set} {T : S -> Set} {s : S} {t t' : T s} -> t == t' -> _ == _ {Sg S T}`  
`{ (s, t) (s, t') } sndq refl = refl`

`uip : {X : 集合} {x y : X} {q q' : x == y} -> q == q' uip {q = refl} {q' = refl} = refl`

`? = ext (\ s -> sndq uip)`

但这有些杀鸡用牛刀，因为唯一的问题只是恼人的等式证明不匹配：实现中可计算的部分完全一致。因此，修复方法是使用 *irrelevance*。我用存在量词替换 `Sg`，其第二个分量用一个点标记为不相关。现在，无论我们使用哪一个证明来表明该见证是好的都无关紧要。

记录 `Ex (S : 集合) (T : S -> 集合) : 集合` 其中构造器 `_,_ 字段 fst`  
`: S .snd : T fst 打开 Ex`

新的候选平衡器是

`Q : {S T : 集合} (f g : S -> T) -> 集合`  
`Q {S} {T} f g = Ex S \ s -> f s == g s`

整个构造照常进行，除了在最后一个义务中

`? = 反射`

被接受了！所以是的，即使在内涵性设定中， $\eta$  定律以及将字段标记为无关的能力也为我们提供了等化子。

在这一构造中未涉及不可判定的类型检查。

### 8.3 我们为什么需要容器？

在我看来，容器（如容器理论中的容器）的价值在于它们的 *uniformity*。这种统一性为将容器表示作为可执行规范的基础提供了相当大的空间，甚至可能用于机器辅助的程序推导。

容器：一种理论工具，而非良好的运行时数据表示策略

我会 *not* 推荐将（归一化）容器的不动点作为实现递归数据结构的一种通用且良好的方式。也就是说，知道某个函子（在同构意义下）可以表示为一个容器是很有帮助的，因为这告诉你：通用的容器功能（由于其统一性，可以一劳永逸地轻松实现）可以被实例化到你的特定函子上，以及你应该期待怎样的行为。但这并不是说容器实现会在任何实际意义上都高效。事实上，我通常更偏好对一阶数据使用一阶编码（标签和元组，而不是函数）。

为了修正术语，我们假设容器的类型 `Cont`（在 `Set` 上，但其他类别也适用）由一个构造器 `<|` 给出，包含两个字段，形状和位置。

S : 集合  
P : S -> 集合

(这是用于确定 Sigma 类型、或 Pi 类型、或 W 类型的同一数据签名，但这并不意味着容器与这些任何一种是同一回事，或这些东西彼此相同。)

诸如函子这样的事物的解释是由

```
[_]C : Cont -> 集合 -> 集合
[S <| P]C X = Sg S \ s -> P s -> X -- 我更倾向于 (s : S) * (P s -> X) mapC : (C : Cont){X Y : Set} -> (
X -> Y) -> [C]C X -> [C]C Y mapC (S <| P) f (s, k) = (s, f o k) -- o 是复合 翻译文本:
```

我们已经在获胜。这就是一劳永逸地实现的映射。更重要的是，函子法则仅通过计算即可满足。构造操作或证明法则时，不需要对类型结构进行递归。

描述是去规范化的容器

没有人试图声称，从操作上讲，Nat <| Fin 提供了一个 *efficient* 列表实现，只是通过做出这个识别，我们学到了一些关于列表结构的有用知识。

让我来说一下 *descriptions*。为了照顾懒惰的读者，让我把它们重建一下。

数据描述: Set1 其中

```
var : Desc sg pi : (A : 集合)(D : A -> Desc) -> Desc one : Desc -- 可能是 Pi 与 A = Z
ero _*_: Desc -> Desc -> Desc -- 可能是 Pi 与 A = 布尔
```

con : 集合 -> 描述 -- 将常量描述作为单例元组 con A = sg A \ \_ -> 一

```
_+_ : 描述 -> 描述 -> 描述 -- 通过与一个标签配对的不交并和 S + T = sg Two \ { true -> S ; false -> T }
```

Desc 中的值描述了其不动点给出数据类型的函子。它们描述的是哪些函子？

```
[_]D : 描述 -> 集合 -> 集合 [ var ]D X = X [ sg A D ]D X = Sg A \
a -> [ D a ]D X [ pi A D ]D X = (a : A) -> [ D a ]D X [ one ]D X =
One [ D * D' ]D X = Sg ([ D ]D X) \ _ -> [ D' ]D X
```

```
mapD : (D : Desc){X Y : Set} -> (X -> Y) -> [ D ]D X -> [ D ]D Y mapD var f x = f x mapD
(sg A D) f (a, d) = (a, mapD (D a) f d) mapD (pi A D) f g = \ a -> mapD (D a) f (g a) mapD
one f <> = <> mapD (D * D' ) f (d, d' ) = (mapD D f d, mapD D' f d' )
```

我们不可避免地需要通过递归处理描述，因此这是一项更艰难的工作。范畴律也不是免费的。操作上我们得到了数据的更好表示，因为我们不需要依赖函数编码，而具体的元组就足够了。但我们必须更加努力地编写程序。

请注意，每个容器都有一个描述：

$\text{sg } S \setminus s \rightarrow \text{pi } (P \ s) \setminus \_ \rightarrow \text{变量}$

但同样也可以说，每个描述都有一个 *presentation* 作为同构容器。

$\text{ShD} : \text{描述} \rightarrow \text{设置 ShD}$   
 $D = [D]D \text{ —}$

$\text{PosD} : (D : \text{描述}) \rightarrow \text{ShD } D \rightarrow \text{设置 PosD 变量 } \langle \rangle = \text{一个 PosD (sg A D) (a, d) = PosD (D a) d PosD (pi A D) f = Sg A \setminus a \rightarrow \text{PosD (D a) (f a) PosD 一个 } \langle \rangle = \text{零 PosD (D * D') (d, d') = PosD D d + PosD D' d ,}$

$\text{ContD} : \text{描述} \rightarrow \text{Cont ContD } D = \text{Sh D D} < | \text{ PosD D}$

也就是说，容器是描述的一种规范形式。可以作为一个练习来证明， $[D]D \ X$  与  $[ContD \ D]C \ X$  自然同构。这使事情变得更容易，因为原则上，要说明对描述该做什么，只需说明对它们的规范形式——容器——该做什么即可。原则上，上述  $\text{mapD}$  操作可以通过将这些同构与  $\text{mapC}$  的统一定义融合而得到。

微分结构：容器指引道路

同样地，如果我们有相等的概念，我们就可以说明容器的单孔上下文是什么 *uniformly*

$\_ - [ \_ ] : (X : \text{Set}) \rightarrow X \rightarrow \text{Set}$   
 $X - [ x ] = \text{Sg } X \setminus x' \rightarrow (x == x') \rightarrow \text{Zero}$

$\text{dC} : \text{Cont} \rightarrow \text{Cont}$

$\text{dC } (S < | P) = (\text{Sg } S \ P) < | (\setminus \{ (s, p) \rightarrow P \ s - [ p ] \})$

也就是说，一个容器中单孔上下文的形状是原始容器形状与孔的位置的组合；这些位置是除孔位置之外的原始位置。这是“乘以指数，递减指数”在微分幂级数时的与证明相关的版本。

这种统一的处理为我们提供了一份规范，由此我们可以推导出用于计算多项式导数的、已有数百年历史的算法。

$\text{dD} : \text{描述} \rightarrow \text{描述 dD 变量} = \text{— dD (sg A D) = sg A \setminus a \rightarrow dD (D a) dD (pi A D) = sg A \setminus a \rightarrow (pi (A - [ a ]) \setminus \{ (a', \_) \rightarrow D a' \}) * dD (D a) dD \text{ —} = \text{构造 零 dD (D * D') = (dD D * D') + (D * dD D')}$

我如何检查我的描述导数算符是否正确？通过将其与容器的导数进行比较！

不要陷入这样的陷阱：仅仅因为对某个想法的呈现并非在操作上有帮助，就认为它在概念上也不可能有帮助。

关于“更自由”的单子

最后一件事。Freer技巧相当于以特定的方式重新排列一个任意函子（切换到Haskell）。

数据  $\text{Obfuncscate } f \ x$  其中

$(:<) :: \text{对于所有 } p. f \ p \rightarrow (p \rightarrow x) \rightarrow \text{Obfuncscate } f \ x$

但是这不是 *alternative* 到容器的转换。这只是容器表示的轻微柯里化。如果我们有 *strong* 存在量词和依赖类型，我们可以写出

数据 Obfuncscate f x 其中

```
(:<) :: pi (s :: exists p. f p) -> (fst s -> x) -> Obfuncscate f x
```

因此，(exists p. f p) 表示形状（你可以选择位置的表示方式，然后用其位置标记每个位置），而 fst 从一个形状中选取存在性的见证（即你所选择的位置表示）。由于它是一个容器式表述，它的优点在于显然是严格正的 *exactly*。

在 Haskell 中，当然，你必须展开存在量词，幸运的是这只留下对类型投影的依赖。正是存在量词的弱性证明了 Obfuncscate f 和 f 的等价性。如果你在具有强存在量词的依赖类型理论中尝试同样的技巧，编码会失去唯一性，因为你可以进行投影并区分不同的表示选择。这就是说，我可以通过以下方式表示 Just 3

仅 () :< const 3

或通过

只是正确 :< \ b -> 如果 b 那么 3 否则 5

而在 Coq 中，例如，这些是可以被证明彼此不同的。

挑战：表征多态函数

每个容器类型之间的多态函数都是以特定方式给出的。那种统一性再次发挥作用，帮助澄清我们的理解。

如果你有一些

```
f : {X : 集合} -> [ S <| T ]C X -> [ S' <| T' ]C X
```

它是（外延）由以下数据给出的，这些数据完全没有提到任何元素：

```
toS : S -> S' fromP : (s : S) -> P' (toS s) -> P s
```

```
f (s , k) = (toS s , k o fromP s)
```

也就是说，定义容器之间的多态函数的唯一方法是说明如何将输入形状转换为输出形状，然后说明如何从输入位置填充输出位置。

对于您首选的严格正值函子表示，给出一个类似紧凑的特征描述，描述消除元素类型的抽象的多态函数。（对于描述，我将准确地使用它们对容器的可约性。）

挑战：捕捉“可转置性”

给定两个函子 f 和 g，说明它们的复合 f o g 很容易：(f o g) x 将 f(g x) 中的内容包装起来，给我们“g 结构的 f 结构”。但是，你能否轻松地强加额外的条件，即所有存储在 f 结构中的 g 结构具有相同的形状？

假设 f >< g 捕捉了 f o g 的 *transposable* 片段，其中所有的 g 形状相同，因此我们也可以将其转化为 f-结构的 g-结构。例如，[] o [] 给出 *ragged* 列表的列表，[] >< [] 给出 *rectangular* 矩阵；[] >< Maybe 给出的是要么全是 Nothing 要么全是 Just 的列表。

给出 ><，用于你偏好的严格正函子的表示。对于容器来说，就是这么简单。

$$(S <| P) >< (S' <| P') = (S * S') <| \{ (s, s') \rightarrow P s * P' s' \}$$

结论

容器以其规范化的先  $\Sigma$  后  $\Pi$  形式，并不旨在成为一种高效的机器数据表示。但了解到某个函子——无论其具体如何实现——可以被呈现为一个容器，有助于我们理解其结构，并为其配备有用的工具。许多有用的构造可以针对容器以抽象的方式一次性给出，而对于其他表示形式则必须逐案给出。因此，是的，了解容器是个好主意，即便只是为了把握你实际实现的那些更具体构造背后的原理。

## 8.4 Applicative/Monad 实例在多大程度上是唯一确定的？

鉴于每个 Applicative 都有一个 Backwards 对应版本，

新类型 `Backwards f x = Backwards {backwards :: f x}` 实例 `Applicative f => Applicative (Backwards f)` 其中 `pure x = Backwards (pure x)` `Backwards ff <*> Backwards fs = Backwards (flip ($) <$> fs <*> ff)`

它是 *unusual* 使得 Applicative 能够被唯一确定，就像（而这与之关系甚远）许多集合以多种方式扩展到单群。

在这个答案中，我设置了一个练习，要求找到至少四个不同的有效 Applicative 实例，用于非空列表：我在这里不会透漏答案，但我会给出一个很大的提示，告诉你如何寻找。

与此同时，在一些精彩的近期工作中（我在几个月前的一次暑期学校上看到过），Tarmo Uustalu 展示了一种相当巧妙的方法来把握这个问题，至少在底层函子是 *container* 的情况下，这里的意义是按照 Abbott、Altenkirch 和 Ghani 的定义。

警告：前方是依赖类型！

什么是容器？如果你手头有依赖类型，你可以一致地给出类似容器的函子  $F$  的表示，将其视为由两个组成部分所决定

1. 一组形状， $S$  : 集合
2. 一个以  $S$  为索引的位置集合， $P : S \rightarrow$  集合

在同构意义下， $F X$  中的容器数据结构由一个依赖对给出：某个形状  $s : S$ ，以及某个函数  $e : P s \rightarrow X$ ，它告诉你位于每个位置上的元素。也就是说，我们定义一个容器的扩展

$$(S <| P) X = (s : S) * (P s \rightarrow X)$$

（顺便一提，如果你把  $\rightarrow$  读作反向幂运算），它看起来很像一个广义幂级数。这个三角形旨在让你联想到一个侧放的树节点，顶点由一个元素  $s : S$  标记，而底边表示位置集  $P s$ 。我们称某个函子为容器，如果它同构于某个  $S <| P$ 。

在 Haskell 中，你可以很容易地得到  $S = F()$ ，但构造  $P$  可能需要相当多的类型技巧。不过那 *is* 是你可以在家里尝试的事情。你会发现，容器在所有常见的多项式类型构造操作下都是封闭的，以及恒等，

$$\text{Id} \sim= () <| \_ \rightarrow ()$$

组成，其中一个整体形状由一个外部形状和每个外部位置的内部形状构成，

$$(S0 <| P0) . (S1 <| P1) \sim= ((S0 <| P0) S1) <| \_ \rightarrow (p0 : P0, P1 (e0 p0))$$

#### 8.4. TO WHAT EXTENT ARE APPLICATIVE/MONAD INSTANCES UNIQUELY DETERMINED?205

和其他一些事情，特别是 *tensor*，其中有一个外形和一个内形（因此“外”与“内”是可以互换的）

$$(S0 <| P0) (X) (S1 <| P1) = ((S0, S1) <| \backslash (s0, s1) \rightarrow (P0 s0, P1 s1))$$

因此  $F(X)G$  表示“具有相同形状的  $G$ -结构的  $F$ -结构”，例如， $[](X)[]$  表示 *rectangular* 列表的列表。不过我跑题了

容器之间的多态函数 每个多态函数

$m$ : 对于所有  $X$ .  $(S0 <| P0) X \rightarrow (S1 <| P1) X$

可以通过 *container morphism* 实现，该 *container morphism* 由两部分以非常特殊的方式构建而成。

1. 一个函数  $f: S0 \rightarrow S1$ ，将输入形状映射到输出形状；
2. 一个函数  $g: (s0: S0) \rightarrow P1 (f s0) \rightarrow P0 s0$ ，将输出位置映射到输入位置。

我们的多态函数是

$$\backslash (s0, e0) \rightarrow (f s0, e0 \cdot g s0)$$

其中输出形状由输入形状计算得到，然后通过从输入位置选取元素来填充输出位置。

（如果你是 Peter Hancock，你会对正在发生的事情有一套完全不同的隐喻。形状是命令；位置是响应；一个容器态射是一个 *device driver*，它将命令向一个方向翻译，再将响应向另一个方向翻译。）

每个容器态射都会给你一个多态函数，但反过来也同样成立。给定这样的一个  $m$ ，我们可以取

$$(f s, g s) = m (s, id)$$

也就是说，我们有一个 *representation theorem*，表示在两个容器之间的每一个多态函数都可以通过这样的  $f, g$  对来给出。

那应用态怎么样？我们在构建这些机制的过程中有点迷失了。不过它 *has* 值得。当单子和应用态的基础函子是容器时，多态函数 `pure` 和 `<*>`，`return` 和 `join` 必须通过相关的容器同态来表示。

让我们先从应用函子开始，使用它们的幺半群表示。我们需要

单位:  $() \rightarrow (S <| P) ()$  乘法: 对于所有  $X, Y$ .  $((S <| P) X, (S <| P) Y) \rightarrow (S <| P) (X, Y)$

用于形状的从左到右映射要求我们交付

$unitS: () \rightarrow S$   $multS: (S, S) \rightarrow S$

所以看起来我们可能需要一个幺半群。当你检查应用法则时，你会发现我们需要 *exactly* 一个幺半群。为容器配备应用结构是 *exactly* 通过合适的位置保持操作来精炼其形状上的幺半群结构。单位没有什么需要做的（因为没有源位置的选择），但对于乘法，我们需要的是每当

$multS (s0, s1) = s$

我们有

$\text{multP}(s0, s1) : P\ s \rightarrow (P\ s0, P\ s1)$

满足适当的恒等性和结合性条件。如果我们转向汉考克的解释，我们正在为命令定义一个单元（跳过，分号），在这里没有办法在选择第二个命令之前查看第一个命令的响应，就像命令是一叠打孔卡片一样。我们必须能够将组合命令的响应拆分成各个命令的独立响应。

因此，形状上的每个单群都给我们提供了一个潜在的应用结构。对于列表，形状是数字（长度），并且有许多单群可以选择。即使形状存在于布尔值中，我们也有相当多的选择。

怎么样 单子？与此同时，对于具有  $M \sim S <| P$  的单子  $M$ 。我们 需要

返回  $\text{Id} \rightarrow M$  连接  $M . M$   
 $\rightarrow M$

首先看形状，这意味着我们需要一种偏斜的单子。

返回  $f : () \rightarrow S$  连接  $f : (S <| P)\ S \rightarrow S \dashv (s : S, P\ s \rightarrow S) \rightarrow S$

它是倾斜的，因为我们在右边得到一堆形状，而不仅仅是一个。如果我们切换到Hancock的解释，我们正在为命令定义一种顺序组合，其中我们确实允许第二个命令根据第一个响应来选择，就像我们在使用电传打字机一样。从几何上讲，我们是在解释如何将树的两层组合成一层。如果这样的组合是唯一的，那将是非常令人惊讶的。

再次，对于位置，我们必须以一致的方式将单一输出位置映射为成对的位置。这对于单子来说更棘手：我们首先选择一个外层位置（response），然后必须选择一个内层位置（response），该内层位置要适合在第一个位置（在第一次 response 之后选定）处发现的形状（command）。

我很想链接到Tarmo的工作以获取详细信息，但似乎它还没有发布。他实际上使用这个分析列举了几种底层容器选择下所有可能的单子结构。我很期待这篇论文！

编辑。为了尊重另一个答案，我应当指出，当处处  $P\ s = ()$  时，那么  $(S <| P)\ X \sim (S, X)$ ，并且单子/应用子的结构彼此之间以及与  $S$  上的幺半群结构完全一致。也就是说，对于 writer 单子，我们只需要选择形状层面的操作，因为在每一种情况下，值都恰好只有一个位置。

## 8.5 观察性类型理论中的模式匹配

我想我来回答这个问题。我觉得这是一个奇怪的问题，但那是因为我个人的经历。简短的回答是：在OTT或任何内核类型理论中不要做模式匹配。这个并不等于永远都不做模式匹配。

长一点的答案基本上就是我的博士论文。

在我的博士论文中，我展示了如何将模式匹配风格编写的高层程序精化为一种内核类型理论，该理论只包含对归纳数据类型的归纳原理以及对命题相等性的恰当处理。模式匹配的精细化会在数据类型索引上引入命题等式，然后通过统一来求解它们。当时我使用的是内涵相等性，但观察相等性至少能提供同样的能力。也就是说：我用于精化模式匹配（从而将其排除在内核理论之外）、并隐藏所有等式层面的杂耍把戏的技术，早于升级到观察相等性。你来说明观点的那种可怕的 vlookup，或许对应于精化过程的输出，但其输入并不一定要那么糟糕。这个漂亮的定义

```
vlookup : Fin n -> Vec X n -> X
vlookup fz (vcons x xs) = x v
lookup (fs i) (vcons x xs) = vlookup i xs
```

可以很好地精化。沿途发生的方程求解，正是 Agda 在通过模式匹配检查定义时在元层面所做的那种方程求解，或者说也是 Haskell 所做的那种。不要被诸如这样的程序所迷惑

```
f :: a ~ b => a -> b
f x = x
```

在 *kernel* Haskell 中，它展开为某种

```
f {q} x = 强制 q x
```

但它不在你面前。而且它也不在编译后的代码中。OTT 等式证明，就像 Haskell 等式证明一样，可以在使用 *closed* 项目进行计算之前被擦除。

*Digression.* 为了明确 Haskell 中相等性数据的地位，GADT

```
数据 Eq :: k -> k -> * 其中 Refl :: Eq x x
```

真的给你

```
Refl :: x ~ y -> 等式 x y
```

但是由于类型系统在逻辑上并不严谨，类型安全依赖于对该类型的严格模式匹配：你不能擦除 *Refl*，并且你确实必须在运行时计算并匹配它，但你可以擦除与  $x \sim y$  的证明对应的数据。在 OTT 中，整个命题片段对于开放项来说是与证明无关的，并且对于闭合计算是可擦除的。  
*End of digression.*

此数据类型上等式的可判定性并不特别相关（至少，如果你有身份证明的唯一性的话；如果你没有 UIP，总有一些方式通过可判定性获得它）。在模式匹配中出现的方程问题是任意的 *open* 表达式。这是相当宽泛的。但机器显然可以决定由变量和完全应用构造器构成的第一阶表达式的片段（这也是 Agda 在分裂案例时所做的：如果约束太奇怪，程序就会报错）。OTT 应该使我们能够进一步推进到高阶统一的可判定片段。如果你知道  $(\text{forall } x. f\ x = t[x])$  对于未知的  $f$ ，这相当于  $f = \lambda x \rightarrow t[x]$ 。

因此，“OTT 中不支持模式匹配”一直都是一个刻意的设计选择，因为我们始终打算把它作为我们已经知道如何进行的某种翻译的展开目标。更确切地说，它是在内核理论能力上的一次严格升级。

## 8.6 OTT 中的可证明相干性

首先，谢谢你提问关于观察类型理论的问题。其次，你在这里所做的 *does* 看起来是连贯的，尽管它的内容与 Thorsten Altenkirch、Wouter Swierstra 和我们在我们版本的故事中放置的位置有所不同。第三，这并不令人惊讶（至少对我来说不惊讶），因为一致性是可以推导出来的，剩下的唯一公设就是反身性。这在我们的 OTT 中也是成立的，而 Wouter 在我们写那篇论文时使用 Agda 1 做了证明。由于证明无关性和生命短暂，我没有将他的证明移植到 Agda 2。

如果你遗漏了什么，它就潜藏在你的评论中

我们仍然需要假定一些东西来定义 *subst* 以及其他内容。

如果你有一些  $P : X \rightarrow$  集合，一些  $a, b : X$  和一些  $q : a = b$ ，你期望得到一个函数在  $P a \rightarrow P b$ 。 “相等的函数对相等的输入产生相等的输出”这个公式给出了这一点，因为  $\text{refl } P : P = P$ ，所以从  $q$  中，我们可以推导出  $P a = P b$ 。你的 “相等的函数对给定的输入产生相等的输出” 这个公式不允许你让  $q$  弥合从  $a$  到  $b$  的差距。

在反射 (*refl*) 和替代 (*subst*) 的存在下，“两个相等的输入”与“一个输入在两个地方使用”是等价的。对我来说，你似乎已经将工作转移到你需要的其他部分以获得替代 (*subst*)。根据你对强制 (*coerce*) 定义的懒惰程度 (以及 *that* 是你获得证明无关性的方式)，你只需要一个公理。

凭借您的特定公式，您甚至可以通过 *homogeneous* 值相等来解决问题。如果您通过强制转换而不是方程式来修复类型间隙，您可能会省去一些麻烦 (也许还能去掉函数相等中的域类型方程)。当然，在这种情况下，您需要考虑如何替代一致性声明。

我们尽力避免在等式的定义中引入强制转换，保持某种对称性，并将类型方程与值方程分开，主要是为了减少一次性需要考虑的内容。有趣的是，至少构造的某些部分可能会因为用 “一个事物及其强制转换” 替代 “两个相等的事物” 而变得更简单。

## 8.7 如何在 Coq 中解决具有无效类型等式的目标?

*tl;dr* 基数论证是唯一能证明类型不相等的方式。你可以通过一些反思更有效地自动化基数论证。如果你想更进一步，可以通过构建一个宇宙为你的类型赋予语法表示，确保你的证明义务是以表示的语法不等式来框定，而不是类型的语义不等式。

### 同构即等式

人们普遍认为 (可能某处已有证明) Coq 的逻辑与公理 *isomorphic* 集合是 *propositionally equal* 一致。实际上，这是弗拉基米尔·沃耶多夫 (Vladimir Voevodsky) 的同伦公理 (Univalence Axiom) 的一种推论，而这个公理目前正引起大家的广泛兴趣。我必须说，在没有类型区分的情况下，它似乎是非常合理的，并且可以构造出一种计算解释，能够通过插入同构中的任一组件，在任何给定时刻将值从相等类型之间传递。

如果我们假设这样的公理是一致的，我们就会发现，在现有的逻辑中，类型不等式只能通过否定类型同构的存在来成立。因此，你的部分解至少在原则上是切中要害的。可枚举性对于证明非同构性至关重要。我不确定  $\text{nat} = (\text{nat} \rightarrow \text{nat})$  的状态如何，但很明显 *from outside the system*， $\text{nat} \rightarrow \text{nat}$  的每一个居住项都有一个正规形，而且正规形的数量是可数的：至少可以认为，存在一些一致的公理或反射原理，使逻辑更加 *intensional*，并且能够验证这一假设。

### 自动化基数参数

我可以看到你可能采取的两步措施，以改善当前的情况。较为温和的一步是通过更好地利用反射来改进你生成这些基数论证的通用技术。你处于理想的位置来做到这一点，因为通常，你的目标是证明一个有限集合与某个更大的集合是不同的。假设我们有一些关于 *DList A* 的概念，这是 *A* 的一个由不同元素组成的列表。如果你能够构造一个 *exhaustive DList A* 和一个 *longer DList B*，那么你就可以证明  $A = B$ 。

有一个 *induction-recursion* 给出的 *DList* 的可爱定义，但 Coq 没有归纳递归。幸运的是，这是一个我们可以通过巧妙使用索引来模拟的定义。请原谅我的非正式语法，但让我们这样写：

参数

A : 集合 d : A -> A -> 布尔 dok : 对所有 x y, d x y = 真 -> x = y -> 假

那就是 d, 代表“不同”。如果一个集合已经具有可判定的相等性, 你可以很容易地为其装备 d。一个大集合可以通过适当的 d 为我们的目的装备, 而不需要太多工作。实际上, 这就是关键步骤: 遵循 SSReflect 团队的智慧, 我们通过使用 bool 而非 Prop 来利用我们领域的“小”, 并让计算机做繁重的工作。

现在, 让我们来

DListBody : (A -> 布尔值) -> 集合

其中索引是 *freshness test* 列表的索引

dnil : DListBody (const true) (\* 任何元素对于空列表来说都是新的 \*) dsnoc : forall f, (xs : DListBody f) -> (x : A) -> is\_true (f x) -> DListBody (fun y => f y  $\wedge$  d x y)

如果你愿意, 你可以存在性地定义 DList 包裹 DListBody。不过, 这可能实际上隐藏了我们想要的信息, 因为要完全展示这样的内容, 步骤如下:

详尽的 (f : A -> bool)(mylist : DListBody f) = 对于所有 x : A, is\_false (f x)

因此, 如果你能为一个有限枚举写出一个 DListBody, 你只需通过带有平凡子目标的情况分析就能证明它是穷尽的。

然后你只需要进行一次鸽巢原理论证。当你想要证明类型之间的不等式时 (假设你已经有合适的 d 候选项), 你穷举较小的集合, 并从较大的集合中展示一个更长的列表, 仅此而已。

在宇宙中工作

更激进的替代方案是质疑你为何一开始就设定这些目标, 以及它们是否真正代表你想要的东西。类型到底应该是什么? 这个问题有多个可能的答案, 但至少可以认为它们在某种意义上是“基数”。如果你希望将类型视为更具体和语法性的, 且只有通过不同的构造方式才能区分它们, 那么你可能需要通过在 *universe* 中工作来为类型提供更具体的表示。你定义了一个“名称”归纳数据类型, 用于表示类型的名称, 并提供解码名称为类型的手段, 然后你将开发框架转变为名称的角度。你应该会发现, 名称的不等式可以通过普通的构造函数区分法来推导。

问题在于, 宇宙构造在 Coq 中可能有些棘手, 再次因为不支持归纳递归。它在很大程度上依赖于你需要考虑的类型。也许你可以归纳地定义一个 U : Set 然后实现一个递归解码器 T : U -> Set。对于简单类型的宇宙, 这当然是可行的。如果你想要一个依赖类型的宇宙, 事情就会有些复杂。至少你可以做到这一点。

U : 类型 (\* 请注意我们已经增加了一个尺寸 \*) NAT : U PI : 对于 (A : 集合), (A -> U) -> U

T : U -> 集合 T NAT = nat T (PI A B) = 对于 (a : A), T (B a)

但请注意,  $\text{PI}$  的定义域在  $\text{Set}$  中是未编码的, 而不是在  $U$  中。归纳-递归的  $\text{Agdans}$  可以克服这一点, 同时定义  $U$  和  $T$

$U$ : 集合 (\* 既好又小 \*)  $\text{NAT} : U$   $\text{PI}$ : 对于任意  $(A : U)$ ,  $(T A \rightarrow U) \rightarrow U$  (\* 注意  $T$  的使用 \*)

$T : U \rightarrow \text{集}$   $T \text{ NAT} = \text{nat } T$   $(\text{PI } A \ B) = \text{对于 } (a : T \ A),$   
 $T \ (B \ a)$

但是  $\text{Coq}$  不会接受这一点。再次说明, 解决方法是使用索引。这里的代价是  $U$  不可避免地很大。

$U$ : 集合  $\rightarrow$  类型  $\text{NAT} : U$   $\text{nat}$   $\text{PI}$ : 对于所有  $(A : \text{集合}) (B : A \rightarrow \text{集合})$ ,  $U \ A \rightarrow$   
 $(\text{对于所有 } a, U \ (B \ a)) \rightarrow U \ (\text{对于所有 } a, B \ a)$

但是你仍然可以用那种方式构建的宇宙完成许多事情。例如, 可以为这样的宇宙装备一个计算上有效的外延相等性。

## 8.8 是否可以在构造演算中表达平衡无标记二叉树的类型?

我不了解  $\text{Morte}$  的细节, 但我对在更广泛的类型化  $\lambda$  演算中可能实现的内容有一些线索。

如果  $\text{Nat}$  被定义为不定式, 可能可以通过迭代定义这些树。

自然数: \* 自然数 =  $(x : *) \rightarrow (x \rightarrow x) \rightarrow x \rightarrow x$  对偶:  
 $* \rightarrow * \rightarrow *$  对偶  $x \ y = (z : *) \rightarrow (x \rightarrow y \rightarrow z) \rightarrow z$  树:  
 $* \rightarrow \text{自然数} \rightarrow *$  树  $a \ n = n * (\lambda t \rightarrow \text{对偶 } t \ t) \ a$

当然, 要做到这一点, 我需要一个 *large* 消除。在这里, 我随意地使用了  $* : *$ , 但一般来说, 这并不安全。归纳定义允许大规模的消除而不会出问题: 不定性编码的数据类型则不然。

但是, 如上所述, 我利用了树的索引结构恰好与索引它们的自然数的索引结构兼容这一事实, 而一般情况下没有理由认为会是这样。索引以各种各样古怪的方式变化: 只有那些表征某种“大小”的索引, 随着我们向内推进而变小。

索引结构确实承认教堂编码表示。只是我们不是在对一个集合进行迭代, 而是在对一个索引集合进行迭代。这里有一种表达方式。

```
Tree : * -> Nat -> *
Tree a n = (x : Nat -> *) ->
            (a -> x Z) ->
            ((n : Nat) -> x n -> x n -> x (S n)) ->
            x n
```

很容易写一些像这样的东西

最左 : (a : \*) -> (n : 自然数) -> 树 a n -> a 最左 a n t = t (\\_ -> a) (\ a -> a) (\\_ -> l)  
l \_ -> l)

但是

leftChild : (a : \*) -> (n : Nat) -> 树 a (S n) -> 树 a n

这是一个更高的要求，需要某种方式来检查或约束数字。这就是为什么 GHC Haskell 有这么多关于相等性的内容，~

## 8.9 在 Coq/Proof General 中进行类似 Agda 的编程？

让我教你一个奇怪的技巧。它可能不是你所有问题的答案，但至少在概念上它可能会有所帮助。

让我们为自然数实现加法，后者定义为

归纳 nat : 集合 := zero : nat | s  
uc : nat -> nat.

你可以尝试用策略来写加法，但会发生这种情况。

定理 plus' : nat -> nat -> nat. 证明.

感应 1.

plus' < 2 个子目标

=====

nat -> nat

子目标 2 是: nat  
-> nat

你看不清自己在做什么。

诀窍在于更仔细地观察你正在做的事情。我们可以引入一个无端的依赖类型，克隆 nat。

归纳 PLUS (x y : nat) : 集合 := | 定义 PLUS : nat -> PLUS x y.

这个想法是，PLUS x y 是“计算 plus x y 的方法”的类型。我们需要一个投影，允许我们提取这种计算的结果。

定理 usePLUS : 对于所有 x y, PLUS x y -> nat. 证明. 归纳 1. 精确  
n. 定义完毕.

现在我们已经准备好通过证明来进行编程。

定理 mkPLUS : 对于所有 x y, PLUS x y. 证明。

mkPLUS < 1 个子目标

=====

对于所有 x y : nat, PLUS x y

目标的结论展示了我们当前的左侧和上下文。在Agda中，C-c C-c的类比是...

感应  $x$ .

mkPLUS < 2 个子目标

```
=====
  对于所有  $y : \text{nat}$ , PLUS zero  $y$ 
```

子目标 2 是: forall  $y : \text{nat}$ , PLUS (suc  $x$ )  $y$

而且你可以看到它已经做了一个情况分割! 我们来处理基础情况。

引入  $y$ 。精确地 (defPLUS zero  $y$   $y$ )。

调用PLUS的构造函数就像写一个方程。想象一下在其第三个参数前面有一个=符号。对于步骤情况, 我们需要进行递归调用。

```
intros  $y$ .
  eapply (fun  $h \Rightarrow$  (defPLUS (suc  $x$ )  $y$       (suc (usePLUS  $x$   $y$   $h$ ))))).
```

为了进行递归调用, 我们调用 usePLUS 并传入我们想要的参数, 这里是  $x$  和  $y$ , 但我们将第三个参数抽象化, 即如何实际计算它的解释。我们剩下的只是那个子目标, 实际上是终止检查。

mkPLUS < 1 子目标

```
 $x$  : 自然数 IH $x$  : 对于  $y$  : 自然数, PLUS  $x$ 
 $y$   $y$  : 自然数
===== P
PLUS  $x$   $y$ 
```

而现在, 你不再使用 Coq 的受保护性检查, 而是使用...

自动.

... 这检查了归纳假设是否覆盖了递归调用。我们

已定义。

我们有一个工作器, 但我们需要一个包装器。

定理 plus : nat -> nat -> nat。证明。intros  $x$   $y$ 。exact  
(usePLUS  $x$   $y$  (mkPLUS  $x$   $y$ ))。已定义。

我们准备好了。

评估计算 (加 (继 (继 零)) (继 (继 零)))。

Coq < = suc (suc (suc (suc zero))) : nat

你 *have* 一个交互式构建工具。你 *can* 游戏它，通过使类型更加信息化，展示你正在解决的问题的相关细节。最终的证明脚本...

定理 `mkPLUS` : 对于任意 `x` 和 `y`, `PLUS x y`。证明。对 `x` 进行归纳。引入 `y`。精确 (`defPLUS zero y` `y`)。引入 `y`。应用 (`fun h => (defPLUS (suc x) y (suc (usePLUS x y h)))`)。自动。定义。

... 明确构建的程序。你可以看到这正在定义加法。

如果你将这个设置自动化以构建程序，然后叠加一个界面来显示你正在构建的程序和关键的简化问题策略，你会得到一个有趣的小编程语言，叫做 `Epigram 1`。