欧阳继超

GROKKING MONAD

函数式装逼手册

Contents

Printable is generated from free softwares: GNU Emacs, orgmode, Tex, Graphivz DOT...

`github.com/jcouyang/grokking-monad`

First printing, 2020

前言

本书的主要目的是为了解释 **为什么需要 *，* 如何理解**以及 **如何使用** Monad，分为三大个部分，猫论/Catergory Theory，食用猫呢/Practical Monads 和搞基猫呢/Advanced Monads。[1]

[1] 什么？你不喜欢谐音梗？我也不喜欢，可是，这也不是讲脱口秀的书啊。~~再说你也就花了六美元，十分适合这种廉价梗~~

猫论/Catergory Theory 是理论基础，解释单子由何而来，若是 ~~不想装逼装得有理有据~~ 觉得太无聊其实可跳过，比较适合好奇心大的猫。

食用猫呢/Practical Monads 提供很多日常会遇到的单子供大家食用。

搞基猫呢/Advanced Monads 适合谁你自然懂得。

其中所有例子都有双语 ~~中文和英语~~ *Haskell* 和 *Scala* 解释，双语例子都成对出现，先 Haskell 后 Scala。

至于为什么选择这两种语言？其实代表两大派系，一个 ML 系一个 Java/C++ 系，若是看不懂 Haskell，Scala 可能会更好懂些。

当掌握了这种思维方式，不局限于 Haskell 或者 Scala，其实可以扩展到任何语言的编程中，即使是没有类型系统的语言，JS，说你呢，不要看人家 PHP。

那么，为啥要花钱买一本不是正规出版社的书？

许多年前，当我还是年少 ~~有为~~ 无知的时候，有个很正规的出版社叫我写过一本书[2]。

[2] `https://book.douban.com/subject/26883736/`

当我听说写书是按字数给钱的时候，我的程序员世界观崩塌了那么一会。什么 DRY[3]，什么 YAGNI[4]，我统统都需要，而且能重复说的概念绝对不能一次说清楚了。

[3] 不要重复 (Don't Repeat Yourself) `https://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

[4] 你可能不会需要的 (You Aren't Gonna Need It) `https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it`

那段时光里 Emacs 那熟练的快捷键 `Ctrl y` 简直就是我的人民币印钞机。

于是我东拼西凑，从我的博客抄了好多字。结果豆瓣才 6.6 分，居然只有几个人说我是抄的，~~真是的，读书人，怎么能说抄呢？~~ 显然读者都不怎么看我的博客。

再说了，~~我抄你家书了吗？~~ 其实书上可比我的博客精致多了，看，我还加了好多萌萌的插画呢。字不算钱，画也多少算点吧。而且，老子写博客的时间不要钱吗？

呵呵，确实不要。

显然第一版还滞销，出版社也再没联系我第二版的版税 的事情。就这么投入一年的时间加工精美博客以获得利润的完美赚钱计划，结果还不如我打工一天赚的钱多。

不过不叫确实好怪我，目标读者是前端，内容显然过于 超前 乏味了些。但是卖不掉我还真是严重怀疑是因为出版社设计的书皮太难看。

你看看那橘色的封面，跟 Emmet[5] 的工服一个颜色，你很难想象这不是一本建筑工程师的书。我要是在图书馆看见这本书，我都怀疑自己走错了到了建筑装潢区，对啊我本来是来借啥书来着？我在哪？我是谁？

[5] Emmet Brickowski：乐高大电影里的一名普通建筑工人。咦，普通工人怎么有 wiki？我都没有 `https://thelegomovie.fandom.com/wiki/Emmet_Brickowski`

你看，这种水平的前言，出版社肯定不乐意要让我改，但是我乐意啊，啊哈哈哈哈。

哦，对，还没说这本书是啥情况呢。其实这本书，它…也是从我的同名博客[6]抄的…

[6] `https://blog.oyanglul.us/grokking-monad/part1`

但其…实我还是有加入更多的内容[7]，更重要的是这个超有设计感的封面。

[7] 比如，你看，博客里就没有这篇前言。

你想想，六美元[8]可以买两瓶可乐，六美元能买一本儿童涂色书，六美元能买二分之一顿饭。你少喝两瓶可乐，让你娃少涂一本涂色书，你少吃半顿饭，买了这本书，把里面的词汇都背下来，在同事领导面前时不时冒出一两个来装逼，升值加薪不是梦。

[8] 啊，那为啥是以美元为单位？那是因为 Gumroad 是按美元给我结算，这样我可以挑选合适的的时候换算成澳元或者人民币花，啊哈哈哈。

或者，打印出来但千万不要装订，抱在怀里从女神男神面前走过时不小心撞一下顺势往天上这么一撒，女神男神在慌乱中一起帮你捡地上散落的书时，肯定会惊叹：哇，这人好厉害居然知道 Monad 是什么？再加上刚才撞一下引起的心跳加速，对你的好感油然而生。

只是少喝两瓶可乐，少涂一本涂色书，少吃半顿饭省下来的六美元[9]，就能提前助你走上人生巅峰的感觉，难道不六吗？

[9] 你看我这写上本书落下的 `Ctrl y` 毛病。

Part I

猫论/Catergory Theory

[10]

> ‘But I don’t want to go among mad people,’ Alice remarked.
>
> ‘Oh, you can’t help that,’ said the Cat: ‘we’re all mad here. I’m mad. You’re mad.’
>
> ‘How do you know I’m mad?’ said Alice. ‘You must be,’ said the Cat, ‘or you wouldn’t have come here.’
>
> Alice didn’t think that proved it at all; however, she went on ‘And how do you know that you’re mad?’
>
> – Alice’s Adventures in Wonderland

> 单子/Monad 是什么？你也不懂, 我也不懂, 我们都不懂.
>
> 话说, 我又怎么知道你不懂呢?
>
> 当然不懂, 不然, 你怎么会来到这里?
>
> 我又是怎么知道自己不懂呢?
>
> 因为，我知道懂的人是什么样子. 显然, 我不是.
>
> 因为，懂的人一定知道猫论/Category Theory.

这一部分主要是纯理论，这里面有很多很装的单词，比如 单子/*Monad*/，它们都是 /斜体，就算一遍没看懂[11]，把这些词背下来也足够装好长一阵子逼了。

这里还有很多代码, 它们都成对出现, 通常第一段是 Haskell, 第二段是 Scala 3.[12]

[10] https://en.wikipedia.org/wiki/Cheshire_Cat

[11] 可以继续看第二部分，看完概念是如何在现实中实现的，再回来看一遍，会感觉好很多。

[12] 为什么用两种语言呢？第一：~~这样代码量会翻倍，可以凑篇幅字数。~~ 这样大家会熟悉多种语言对同一概念的诠释，从而举一反三。第二：读者受众会大一点，因为毕竟 Haskell 的表述比较简洁，有可能很容易理解，但是跟主流语言的表达方式大为不同，也有可能很难适应，加上表达方式更为具体的 Scala，便于加深理解。

1
范畴/*Category*

对于计算机科学的学生，范畴并不是一个新的概念，在本科大纲里，大家都应该学过 离散数学（*Discrete Mathematics*），其中会讲很多 集合论（*Set Theory*）图论，抽象代数的东西。现在回头看看，其实也就是集合论和抽象代数的内容。

所以下面的概念都点到为止，只为解释写成代码会长什么样。

一个 范畴/*Category* 包含两个玩意:

- 东西 `O` （Object）
- 两个东西的关系，箭头 `~>` （态射/*Morphism* ）

还必须带上一些属性:

- 一定有一个叫 id 的箭头，也叫做 1
- 箭头可以 组合 compose/

恩, 就是这么简单!

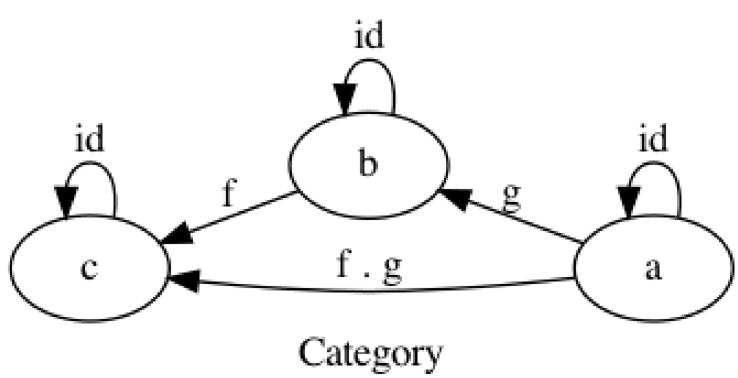


Figure 1.1: 有东西 a, b, c 和箭头 f, g 的 Category，其中 f . g 表示 compose f 和 g

注意到为什么我会箭头从右往左，接着看代码, 你会发现这个方向跟 compose 的方向刚好一致!

这些玩意对应到 Haskell 的 Typeclass 大致就是这样:

```
class Category (c :: * -> * -> *) where
  id :: c a a
  (.) :: c y z -> c x y -> c x z
```

Listing 1.1: Category definition in Haskell

如果这是你第一次见到 Haskell 代码，没有关系，语法真的很简单：

- `class` 定义了一个 TypeClass，`Category` 是这个 TypeClass 的名字
- Type class 类似于定义类型的规范，规范为 `where` 后面那一坨
- 类型规范的对象是参数 `(c:: * -> * -> *)`, `::` 后面是 c 的类型
- c 是 *higher kind* `* -> *`，跟 higher order function 的定义差不多，它是接收类型，构造新类型的类型。这里的 c 接收一个类型，再接收一个类型，就可以返回个类型。
- `id:: c a a` 表示 c 范畴上的 a 到 a 的箭头
- `.` 的意思 c 范畴上，如果喂一个 y 到 z 的箭头，再喂一个 x 到 y 的箭头，那么就返回 x 到 z 的箭头。

而 Scala 可以用 trait 来表示这个 typeclass:

```
trait Category[C[_, _]] {
  def id[A]: C[A, A]
  def <<<(a: C[Y, Z], b: C[X, Y]): C[X, Z]
}
```

Listing 1.2: Category definition in Scala

如果这是你第一次见到 Scala 代码，没关系，从 Haskell 可以飞快的切换过来：

- `class` -> `trait`

=c * -> * -> *= -> `C[_, _]`

- `::` -> `:`
- 函数名前加 `def`

另外 compose 在 haskell 中直接是句号 `.`
scala 中用习惯用 `<<<` 或者 `compose`
总之，我们来用文字再读一遍上面这些代码就了然了.
范畴 C 其实就包含：

1. 返回 A 对象到 A 对象的 id 箭头
2. 可以组合 Y 对象到 Z 对象和 X 对象到 Y 对象的箭头 compose

简单吧？还没有高数抽象呢。

1.1 Hask

Haskell 类型系统范畴叫做 Hask。

在 Hask 范畴上：

- 东西就是类型
- 箭头是类型的变换，即 `->`
- id 就是 id 函数的类型 `a -> a`
- compose 当然就是函数组合的类型

```
type Hask = (->)
instance Category (Hask:: * -> * -> *) where
  id a = a
  (f . g) x = f (g x)
```

我们看见新的关键字 `instance` ，这表示 Hask 是 Type class Category 的实例类型，也就是说对任意 Hask 类型, 那么就能找到它的 id 和 compose

```
given Category[=>[_, _]] {
  def id[A]: A => A = identity[A]
  def <<<[X, Y, Z](a: Y => Z, b: X => Y) = a compose b
}
```

Scala 中, 只需要 new 这个 trait 就可以实现这个 typeclass

其中: identity `Hask a a` 就是

```
(->) a a -- or
a -> a -- 因为 -> 是中缀构造器
```

```
A => A
```

1.2 Duel

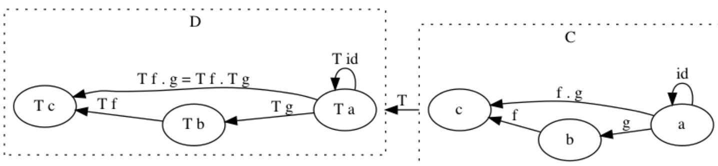
每个 Category 还有一个镜像，什么都一样，除了箭头是反的。

2

函子 / *Functor*

两个范畴中间可以用叫 Functor 的东西来连接起来，比如一个从范畴 C 到范畴 D 的函子 T，我们可以标作 `Functor C D T`。

#+Functor Category



Functor C D T

所以大部分把函子或者单子比喻成盒子其实在定义上是错的，虽然这样比喻比较容易理解，在使用上问题也不大。但是，函子只是从一个范畴到另一个范畴的箭头而已。

- 范畴间东西的函子标记为 `T(O)`
- 范畴间箭头的函子标记为 `T(~>)`
- 任何范畴 C 上存在一个 T 把所有的 O 和 ~> 都映射到自己，标记为函子 1_C
  - $1_C(O) = O$
  - $1_C(\sim>) = \sim>$

```haskell
class (Category c, Category d) => Functor c d t where
  fmap :: c a b -> d (t a) (t b)
```

Listing 2.1: 函子的 Haskell 定义

```scala
trait Functor[C[_, _], D[_, _], T[_]]:
  def fmap[A, B](c: C[A, B]): D[T[A], T[B]]
```

Listing 2.2: 函子的 Scala 定义

`Functor c d t` 这表示从范畴 c 到范畴 d 的一个 Functor t

如果把范畴 c 和 d 都限制到 Hask 范畴：

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

```
trait Functor[=>[_, _], =>[_, _], T[_]]:
 def fmap[A, B](c: =>[A, B]): =>[T[A], T[B]]
```

-> 或者 => 可以写在中间的:

这样就会变成我们熟悉的函子定义：[1]

[1] 这里可以把 Functor 的第一第二个参数消掉, 因为已经知道是在 Hask 范畴了

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

```
trait Functor[T[_]]:
  def fmap[A, B](c: A => B): T[A] => T[B]
```

而 自函子/*endofunctor* 就是这种连接相同范畴的 Functor，因为它从范畴 Hask 到达同样的范畴 Hask。

这回看代码就很容易对应上图和概念了, 这里的自函子只是映射范畴 `->` 到 `->`, 箭头函数那个箭头, 类型却变成了 `t a` 。

这里的 fmap 就是 T(~>)，在 Hask 范畴上，所以是 T(->), 这个箭头是函数，所以也能表示成 T(f) 如果 `f:: a -> b`

3

Cat/猫

递归的, 当我们可以把一个范畴看成一个对象，函子看成箭头的话，那么我们又得到了一个新的范畴，这种对象是范畴箭头是函子的范畴我们叫它 – Cat/猫。

已经没/meow 的办法用语言描述这么高维度的事情了，请回忆并把 C 和 D 想象成点。

4 自然变换 / *Natural Transformations*

函子是范畴间的映射，所以如果我们现在又把 Cat 范畴看成是对象, 那 Cat 范畴之间的箭头，其实就是函子的函子，又升维度了，我们有个特殊的名字给它，叫 ~~喵的变换~~ 自然变换/*Natural Transformations* 。

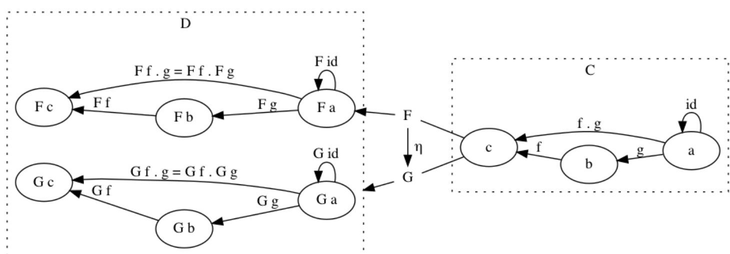


Figure 4.1: Functor F 和 G 以及 F 到 G 的自然变化

范畴 c 上的函子 f 到 g 的自然变化就可以表示成:

```
type Nat c f g = c (f a) (g a)
```

Scala 3 的 rank n types[1] 也很简洁:

```
type Nat[C[_,_],F[_],G[_]] = [A] => C[F[A], G[A]]
```

如果换到 Hask 范畴上的自然变化就变成了:

```
type NatHask f g = f a -> g a
```

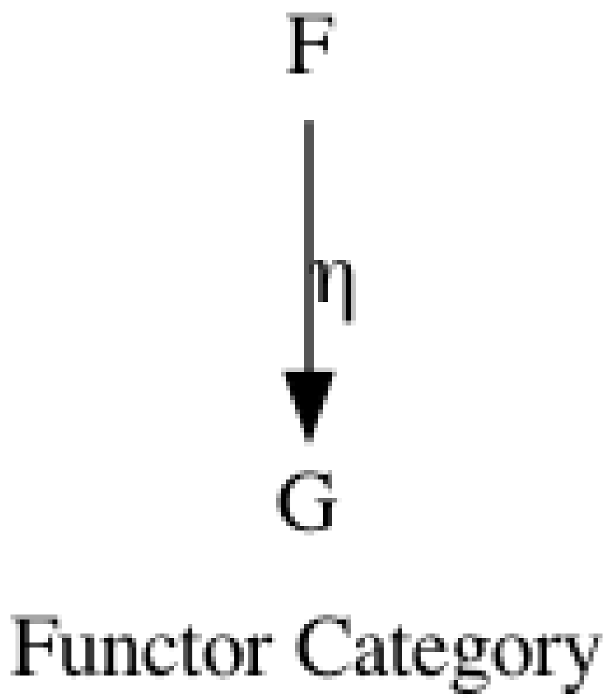
```
type Nat[F[_],G[_]] = [A] => F[A] => G[A]
```

这就是 Scala 中常见的 FunctionK[2]。
恭喜你到达 Functor 范畴.
当然, 要成为范畴，还有两个属性:

- id 为 f a 到 f a 的自然变换
- 自然变换的组合

[1] `https://blog.oyanglul.us/scala/dotty/en/rank-n-type` 别急, 后面马上讲到

[2] `https://blog.oyanglul.us/scala/dotty/en/functionk`



别着急, 我们来梳理一下，如果已经不知道升了几个维度了，我们假设类型所在范畴是第一维度

- 一维：Hask，东西是类型，箭头是 ->
- 二维：Cat，东西是 Hask，箭头是 Functor
- 三维：Functor 范畴，东西是 Functor，箭头是自然变换

感觉到达三维已经是极限了，尼玛还有完没完了，每升一个维度还要起这么多装逼的名字，再升维度老子就画不出来了。

所以，是时候引入真正的技术了 – String Diagram。

5
String Diagram

String Diagram[1] 的概念很简单，就是点变线线变点。

还记得当有了自然变换之后，三个维度已经没法表示了，那原来的点和线都升一维度，变成线和面，这样，就腾出一个点来表示自然变换了。

[1] `https://www.youtube.com/watch?v=kiXjcqxVogE&list=PL50ABC4792BD0A086&index=5`

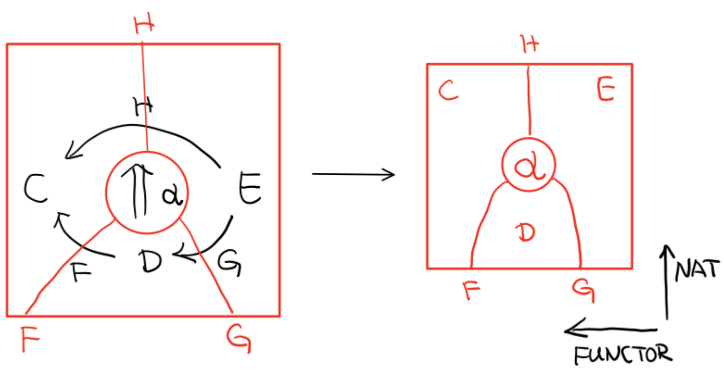


Figure 5.1: String Diagram：自然变换是点，函子是线，范畴是面，自然变换是点

组合（compose）的方向是从右往左，从下到上。

阅读起来，你会发现左右图给出的信息是完全等价的：

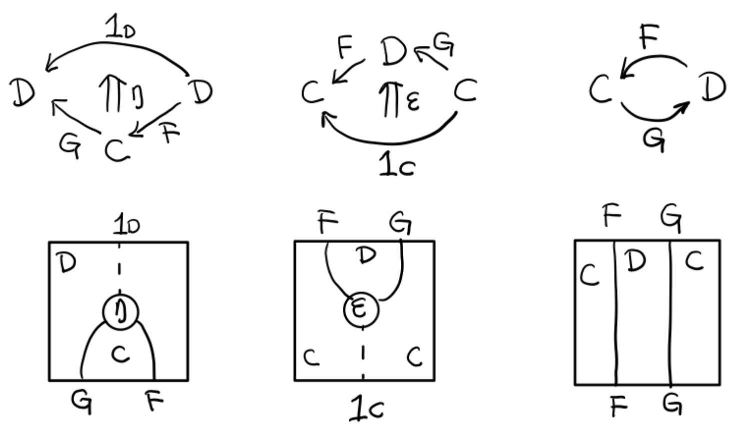
1. 范畴 E 通过函子 D 到范畴 D，范畴 D 通过函子 F 到范畴 C
2. 范畴 E 通过函子 E 到范畴 C
3. F . G 通过自然变换 α 到 H

6

Adjunction Functor 伴随函子

伴随函子是范畴 C 和 D 之间有来有回的函子，为什么要介绍这个，因为它直接可以推出单子。

让我们来看看什么叫有来回。



其中：

- 图右：一个范畴 C 可以通过函子 G 到范畴 D，再通过函子 F 回到 C，那么 F 和 G 就是伴随函子。
- 图中：范畴 C 通过函子组合 F . G 回到范畴 C，函子 G . F 通过自然变换 η 到函子 1_D
- 图左：范畴 D 通过函子组合 G . F 回到范畴 D，函子 1_C 通过自然变化 ϵ 到函子 F . G

同时根据同构的定义，G 与 F 是 同构的。

同构指的是若是有

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

那么 f 与 f’ 同构，因为 `f . f' = id = f' . f`

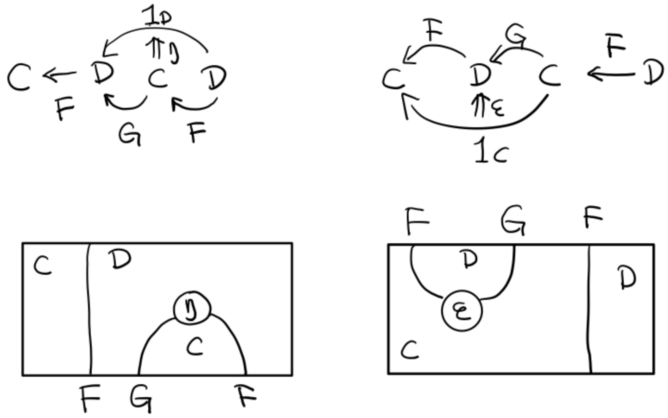


Figure 6.1: 伴随函子的两个 Functor 组合, 左侧记为 F eta, 右侧记为 epsilon F

伴随函子的 F . G 组合是 C 范畴的 id 函子 `F . G = 1_c`

注意看坐标，该图横着组合表示函子组合，竖着是自然变换维度，因此是自然变换的组合。

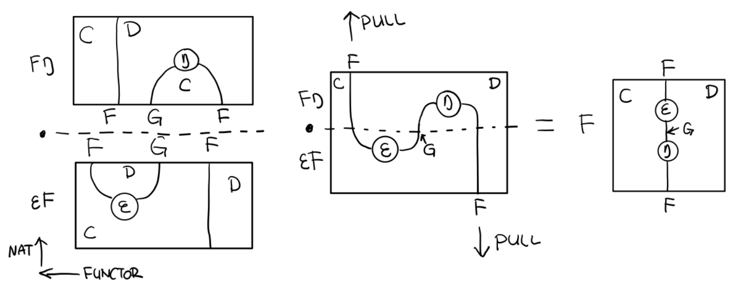


Figure 6.2: eta . epsilon = F -> F

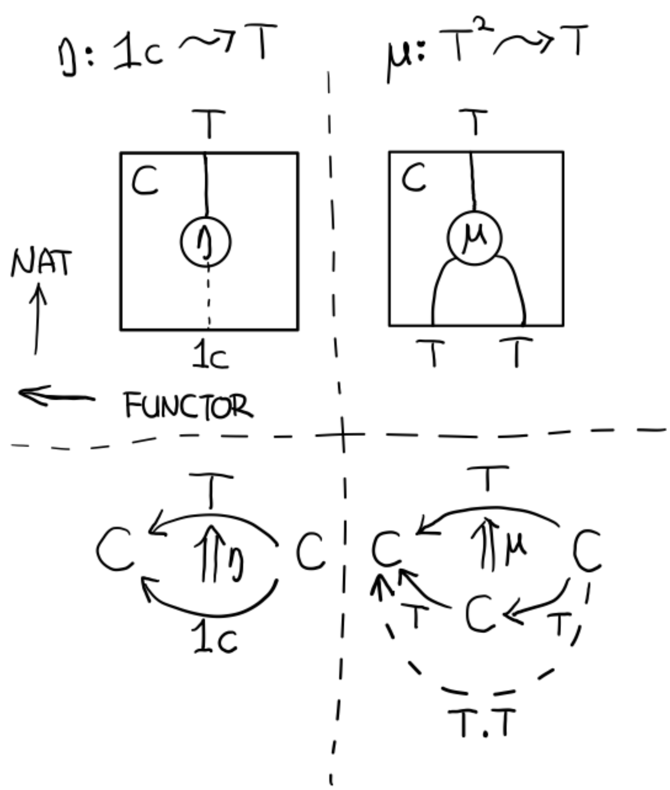
当组合两个自然变换 η . ϵ 得到一个弯弯曲曲的 F 到 F 的线时，我们可以拽着 F 的两端一拉，就得到了直的 F 线。

String Diagram 神奇的地方是所有线都可以拉上下两端，因为线不管是弯的还是直的，包含的信息并不会发生变化。这个技巧非常有用，在之后的单子推导还需要用到。

7
从伴随函子到单子/*Monad*

有了伴随函子，很容易推出单子，让我们先来看看什么是单子：

- 首先，它是一个自函子（endofunctor）T
- 有一个从 i_c 到 T 的自然变化 η (eta)
- 有一个从 T^2 到 T 的自然变化 μ (mu)




```
class Endofunctor c t => Monad c t where
  eta :: c a (t a)
  mu  :: c (t (t a)) (t a)
```

```
trait Monad[C[_, _], T[_]]] extends Endofunctor[C, T]:
  def eta[A]: C[A, T[A]]
  def mu[A]: C[T[T[A]], T[A]]
```

同样，把 c = Hask 替换进去，就得到更类似我们 Haskell 中 Monad 的定义

```
class Endofunctor m => Monad m where
  eta :: a -> (m a)
  mu :: m m a -> m a
```

```
trait Monad[M[_]] extends Endofunctor[M]:
  def eta[A]: A => M[A]
  def mu[A]: M[M[A]] => M[A]
```

要推出单子的 η 变换，只需要让 FG = T。可以脑补一下，因为是自函子，因此可以抹掉 D，想象一下，当 D 这一块面被拿掉之后，线 F 和线 G 是不是就贴在一起了呢？两根贴着的线，不就是一根线吗？

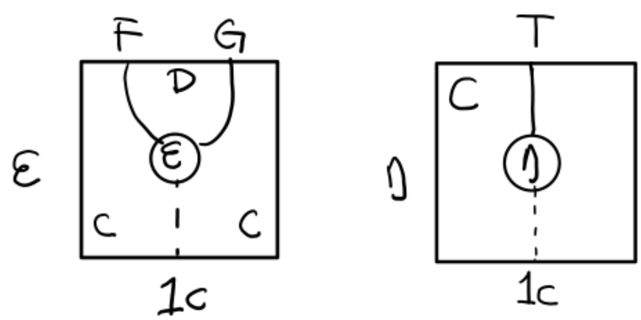


Figure 7.1: 伴随函子的 epsilon 就是单子的 eta

同样的，当 FG = T, 也就是把 D 这陀给抹掉，F 和 G 就变成了 T。

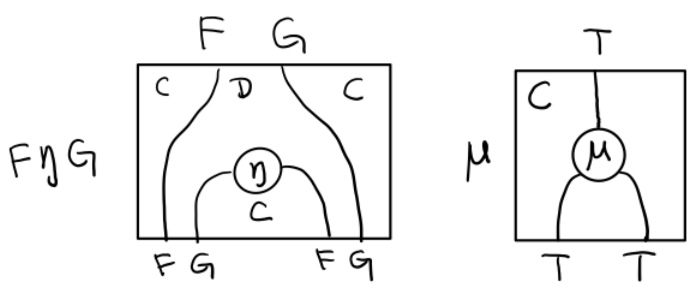


Figure 7.2: 伴随函子的 F eta G 是函子的 mu

7.1 三角等式

三角等式是指 μ . T η = T = μ . η T

要推出三角等式只需要组合 F η G 和 ϵ F G

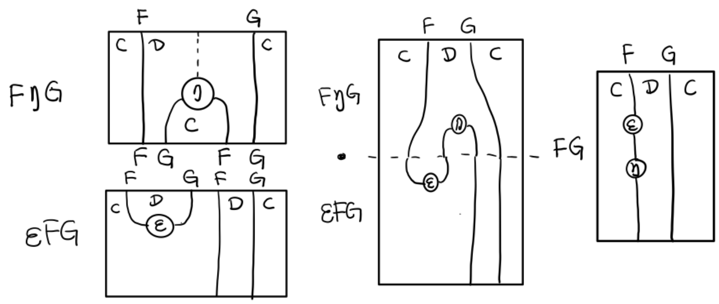


Figure 7.3: F eta G . epsilon F G = F G

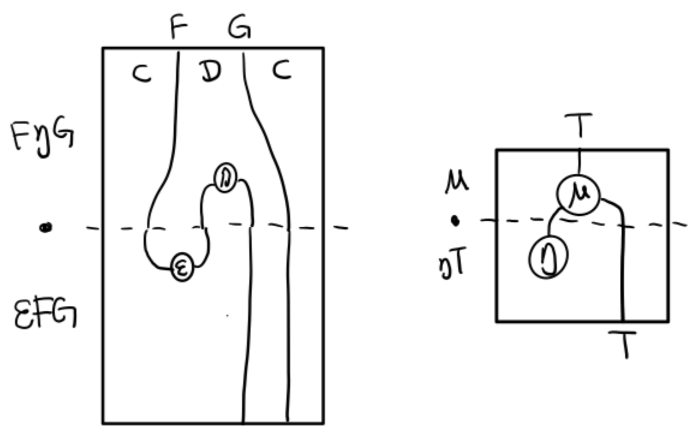


Figure 7.4: F eta G . epsilon F G= F G 对应到 Monad 就是 mu . eta T = T

换到代码上来说

```
(mu . eta) m = m
```

同样的，左右翻转也成立

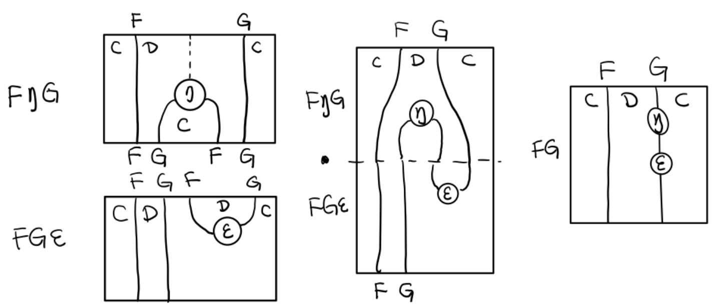


Figure 7.5: F eta G . F G epsilon = F G

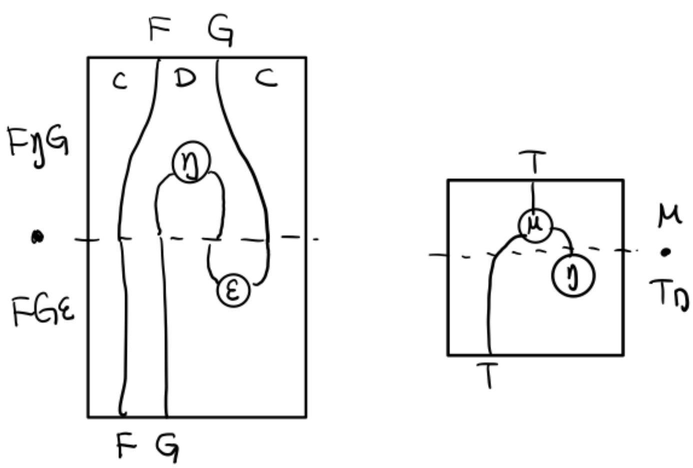


Figure 7.6: F eta G . F G epsilon = F G 对应到 Monad 是 mu . T eta = T

T η 就是 fmap eta

```
(mu . fmap eta) m = m
```

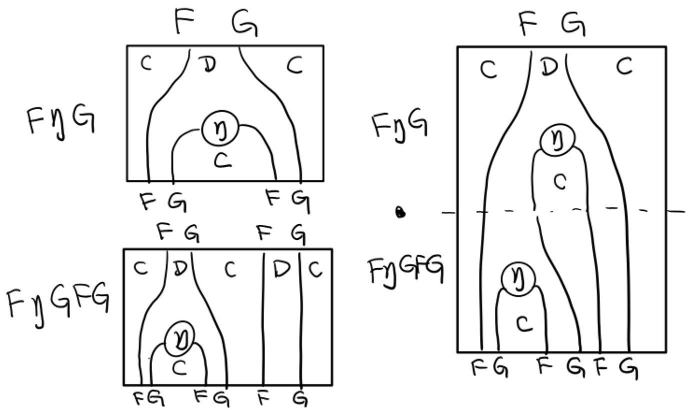
如果把 `mu . fmap` 写成 `>>=`，就有了

```
m >>= eta = m
```

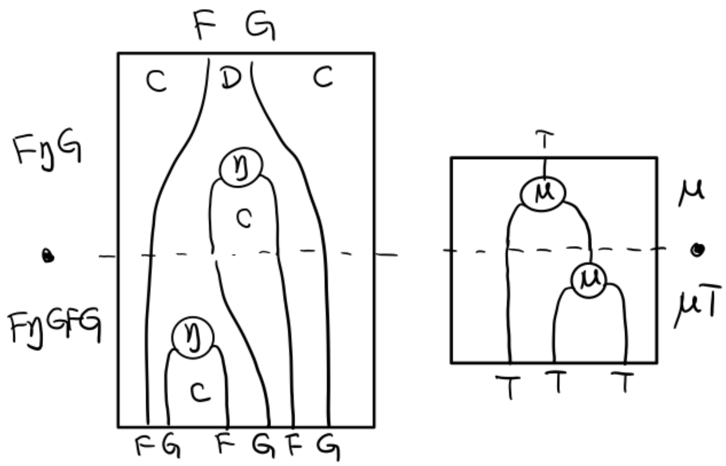
7.2 结合律

单子另一大定律是结合律，让我们从伴随函子推起

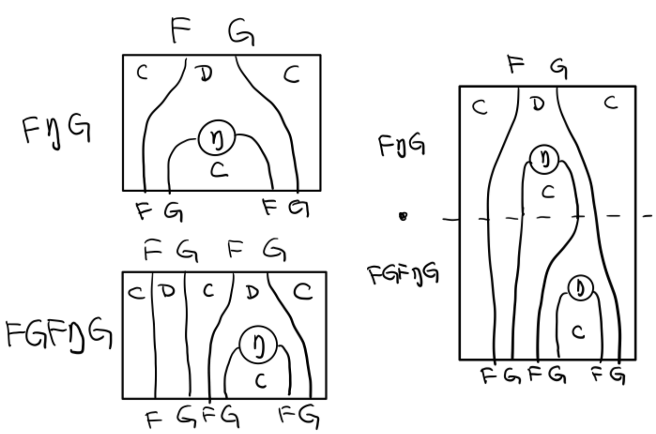
假设我们现在有函子 F η G 和函子 F η G F G, compose 起来会变成 F η G . F η G F G



用 F G = T ，F η G = μ 代换那么就得到了单子的 μ . μ T



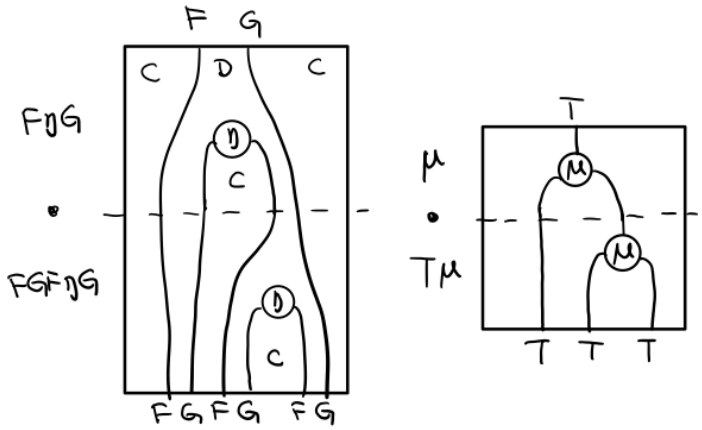
当组合 F η G 和 F G F μ G 后，会得到一个镜像的图



对应到单子的 μ . T μ

结合律是说 $\mu \,.\, \mu \mathrm{T} = \mu \,.\, \mathrm{T} \mu$，即图左右翻转结果是相等的，为什么呢？看单子的 String Diagram 不太好看出来，我们来看伴随函子

如果把左图的左边的 μ 往上挪一点，右边的 μ 往下挪一点，是不是跟右图就一样了



结合律反映到代码中就是

```
mu . fmap mu = mu . mu
```

代码很难看出结合在哪里，因为正常的结合律应该是这样的 (1+2)+3 = 1+(2+3)，但是不想加法的维度不一样，这里说的是自然变换维度的结合，可以通过 String Diagram 很清楚的看见结合的过程，即 μ 左边的两个 T 和先 μ 右边两个 T 是相等的。

8

Yoneda lemma / ~~米田共~~ 米田引理

米田引理是说所有的函子 `f a` 一定存在两个变换 `embed` 和 `unembed=`，使得 `=f a` 和 `(a -> b) -> F b` 同构。

要再 Haskell 中做到这一波操作需要先打开 `RankNTypes` 的编译器开关：

```
{-# LANGUAGE RankNTypes #-}

embed :: Functor f => f a -> (forall b . (a -> b) -> f b)
embed x f = fmap f x

unembed :: Functor f => (forall b . (a -> b) -> f b) -> f a
unembed f = f id
```

Scala 3 不需要插件或者开关[1]，如果是 Scala 2 可以用 `apply` 来模拟. 比如 Cats 中 FunctionK(~>)。

[1] `https://blog.oyanglul.us/scala/dotty/rank-n-type`

```
type ~>[F[_],G[_]] = [A] => F[A] => G[A]
def embed[F[_], A](fa: F[A])(using F: Functor[F]) =
  [B] => (fn: A=>B) => f.fmap(fn)(fa)
def unembed[F[_]](fn: [B] => (A => B) => F[B]): F[A] =
  fn(identity)
```

`embed` 可以把 `f a` 变成 `(a -> b) -> f b`

`unembed` 是反过来，`(a -> b) -> f b` 变成 `f a`

上个图可能就明白了：

这个引理看似很巧妙，特别是用 id 的这个部分，但是有什么用呢？

如果着急可以跳到 Free Monad/自由单子部分，你会发现他是自由单子的基础。而且如果再往后会介绍的宇宙本原左看和右看，更会发现其中得精妙相似之处。

8.1 Rank N Type

前面说好的要解释 Rank N Type，这里赶快补充一下，不然等会我就忘了。

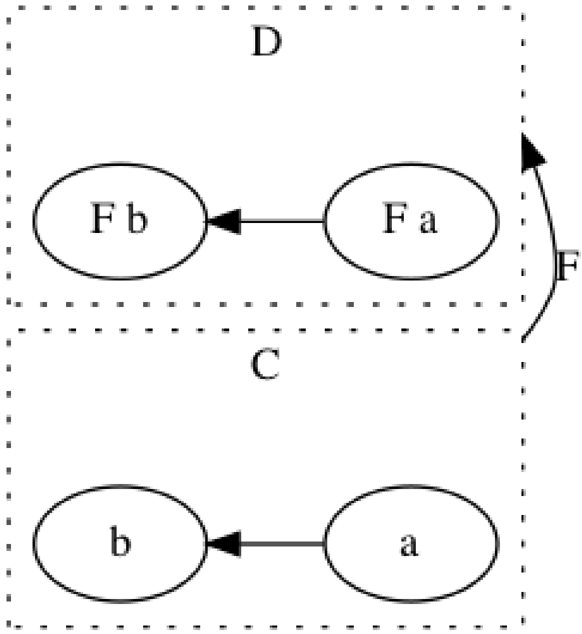


Figure 8.1: 也就是说，图中无论知道 a->b 再加上任意一个 F x，都能推出另外一个 F

Haskell 中可以不用声明类型, 但是其实是省略掉 universally quantified `forall`, 如果把 forall 全部加回来, 就明了很多:

- Monomorphic Rank 0 / 0 级单态[2]: t
- Polymorphic Rank 1 / 1 级 变态 多态: forall a b. a -> b
- Polymorphic Rank 2 / 2 级多态: forall c. (forall a b. a -> b) -> c
- Polymorphic Rank 3 / 3 级多态: forall d . (forall c . (forall a b . a -> b) -> c) -> d

[2] 也就不是不变态

看 rank 几只要数左边 forall 的个数就好了.
一级多态只锁定一次类型 a 和 b
二级多态可以分两次确定类型, 第一次确定 c, 第二次确定 a b
三级多台分三次: 第一次 d, 第二次 c, 第三次 a b
比如:

```
rank2 :: forall b c . b -> c -> (forall a. a -> a) -> (b, c)
rank2 b c f = (f b, f c)

rank2 True 'a' id
-- (True, 'a')
```

- `f` 在 `f True` 时类型 `Boolean -> Boolean` 是符合 `forall a. a->a` 的

- 与此同时 `f 'a'` 时类型确实是 `Char -> Char` 但也符合 `forall a. a->a`

看 Scala 的更简单，因为 Scala 不能省去 universally quantified，只需要数方括号即可。最左边 `[B, C]` 是 rank1，`fn` 的类型里的 `[A]` 是 rank2。

```
def rank2[B, C](b: B, c: C)(fn: [A] => A => A): (B, C) =
  (fn(b), fn(c))

rank2(true, 'a')([A] => (a: A) => A)
```

如果不用 rank2 而是只有 rank1 类型系统就懵逼了:

```
rank1 :: forall a b c . b -> c -> (a -> a) -> (b, c)
rank1 b c f = (f b, f c)
```

```
def rank1[A, B, C](b: B, c: C)(fn: A => A): (B, C) =
  (fn(b), fn(c))
```

f 在 `f True` 是确定 a 是 Boolean，在 rank1 多态是时就确定了 `a -> a` 的类型一定是 `Boolean -> Boolean` ，然后当看到 `f 'a'` 时类型就挂了，因为 `'a'` 不是 `Boolean` 。

9

Kleisli Catergory

函子/Functor 的范畴叫做函子范畴/Functor Catergory, 自然变换是其箭头。那单子/Monad 也可以定义一个范畴吗?[1]

是的, 这个范畴名字叫做 ~~单子范畴~~[2] 可莱斯利范畴/Kleisli Catergory[3]，那么 Kleisli 的箭头是什么?

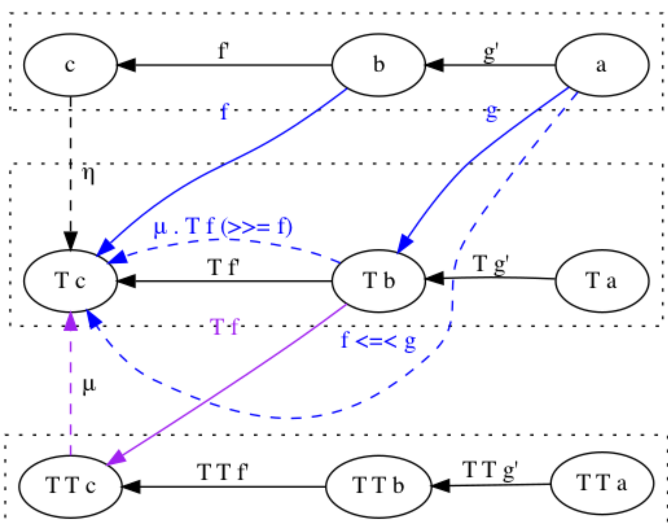


Figure 9.1: 注意观察大大箭 <=< 的轨迹，不知道 dot 为什么会把这根线搞这么又弯又骚的，和 »= 。所以 Kleisli 其实就是斜着走的一个范畴，但是 »= 把它硬生生掰 弯 直了。

我们看定义，Kleisli Category：

1. 箭头是 Kleisli 箭头 `a -> T b`

2. 东西就是 c 范畴中的东西. 因为 a 和 b 都是 c 范畴上的，由于 T 是自函子，所以 T b 也是 c 范畴的

看到图上的 T f/ fmap f 和 μ 了没? [4]

```
f :: b -> T c
fmap f :: T b -> T T c
mu :: T T c -> T c
```

[1] 当然, 单子是自函子，所以也可以是自函子范畴

[2] 怎么说也是函数式编程的核心, 怎么可以叫的这么 low 这么直接

[3] 这个是我瞎翻译的, 但是读出来就是这么个意思，真的，不骗你，照这么读绝对装的一手好逼，不会被嘲笑的

[4] (敲黑板) 就是紫色那根嘛!

```
def f[T[_], B, C](b: B): T[C]
def fmap[T[_], B, C](f: B => C)(tb: T[B]): T[T[C]]
def mu[T[_], C](ttc: T[T[C]]): T[C]
```

紫色的箭头 `T f` [5] 和紫色的虚线箭头 μ 连起来就是 `T f'`, 那么最出名的 bind `>>=` 符号终于出来了:

[5] 即 fmap f

```
tb >>= f = (mu . fmap f) tb
```

Scala 中通常叫作 `flatMap` ，但如果你用 Cats 也是可以用 `>>=` 的。

```
def flatMap[T[_], B, C](f: B => T[C])(tb: T[B]): T[C] = (mu compose fmap(f))(tb)
```

下面这个大火箭 `<=<` 可以把蓝色箭头组合起来.

```
(f <=< g) = mu . T f . g = mu . fmap f . g
```

```
def <=<[T[_], A, B, C](f: B => T[C])(g: A => T[B]): A => T[C] =
  mu compose fmap(f) compose g
```

因此大火箭就是 Kleisli 范畴的 `compose`

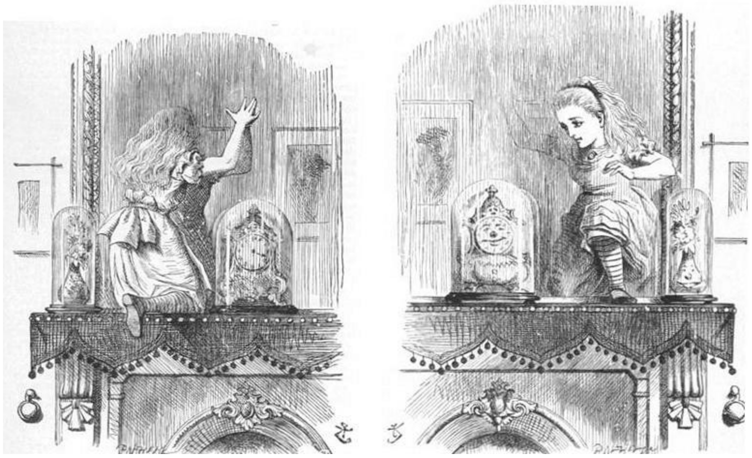
```
(<=<) :: Monad T => (b -> T c) -> (a -> T b) -> (a -> T c)
```

10
Summary

第一部分理论部分都讲完了，如果你读到这里还没有被这些吊炸天/乱七八糟的概念劝退，那么你这份如此强大得信念感，其实到后面两部分也不会有什么用。因为，接下来的例子会很简单，我们要通过编程中常遇到的场景看看理论到底该如何得到实践?

Part II

食用猫呢/Practical Monads



11

Functor 食用函子定义

```haskell
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
  (<$) :: a -> f b -> f a
  (<$) =  fmap . const
```

```scala
trait Functor[F[_]]:
  def fmap[A, B](fn: A => B): F[A] => F[B]
  extension [B](fb: F[B])
    def <$(a: A)(fb: F[B]): F[A] = fmap(const(a))
```

12 Applicative

```haskell
class Functor f => Applicative f where
  pure       :: a -> f a
  (<*>)      :: f (a -> b) -> f a -> f B
  liftA2     :: (a -> b -> c) -> f a -> f b -> f c
  liftA2 f x = (<*>) (fmap f x)
  (*>)       :: f a -> f b -> f b
  a1 *> a2   = (id <$ a1) <*> a2
  (<*)       :: f a -> f b -> f a
  (<*)       = liftA2 const
```

```scala
trait Applicative[F[_]] extends Functor[F]:
  def pure[A](a: A): F[A]
  def ap[A, B](fab: F[A=>B]): F[A] => F[B]
  def liftA2[A, B, C](f: A => B => C): F[A] => F[B] => F[C] = (x: F[A]) =>
    ap(fmap(f)(x))
  extension [A](fa: F[A])
    def *>(fb: F[B]) = (identity <$ fa) <*> fb
    def <*(fb: F[B]) = liftA2(const)
  extension [A, B](fab: F[A => B])
    def <*>(fa: F[B]) = ap(fab)(fa)
```

13 Monad

```haskell
class Applicative m => Monad m where
 (>>=)        :: forall a b. m a -> (a -> m b) -> m b
 (>>)         :: forall a b. m a -> m b -> m b
 m >> k       = m >>= \_ -> k
 return       :: a -> m a
 return       = pure
```

```scala
trait Monad[M[_]] extends Applicative[M] {
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  extension [A, B](ma: M[A])
    def >>(mb: M[B]): MB = flatMap(ma)((_: A) => mb)
}
```

14
Identity 本身就有

本身就有单子/ Identity Monad[1] 可能是最简单的单子了。本身不包含任何计算, 且只有一个构造器:

```haskell
newtype Identity a = Identity { runIdentity :: a }
```

```scala
case class Identity[A](run: A)
```

- 这里取名 `Identity` 叫 **本身就有**，所以 `Identity a` 就是 **本身就有 a**
- 这里使用 `newtype` 而不是 `data` 是因为 `Identity` 与 `runIdentity` 是 同构的[2].

```haskell
Identity :: a -> Identity a
runIdentity :: Identity a -> a
```

你看 `runIdentity . Identity = id` ，所以他们是同构的。

左边的 `Identity` 是 类型构造器[3], 接收类型 `a` 返回 `Identity a` 类型。

如果 `a` 是 `Int`, 那么就得到一个 `Identity Int` 类型。

右边的 `Identity` 是数据构造器，也就是构造值，比如 `Identity 1` 会构造出一个值，其类型为 `Identity Int` 。

大括号比较诡异，可以想象成给 `a` 自动生成了一个 `Identity a -> a` 的函数，比如:

```haskell
runIdentity (Identity 1)
```

```scala
Identity(1).run
```

会返回 1

本身就有可以实现 Functor 和 Monad，就得到 Identity 函子和 Identity 单子。

```haskell
instance Functor Identity where
  fmap f (Identity a) = Identity (f a)
```

[1] 从来没见过有人给这些数据类型按过中文名字, 不然我来, 这样也更好的体会这些数据类型的意图.

[2] 见 第一部分伴随函子

[3] 也就是 Kind * -> *, 因为它非常的 nice, 一定要等到 a 才出类型

```haskell
instance Monad Identity where
  return a = Identity a
  Identity a >>= f = f a
```

而 Scala 则用 `given` 来实现 typeclass:

```scala
given Functor[Identity]:
  def fmap[A, B](f: A => B): Identity[A] => Identity[B] =
    case Identity(a) => Identity(f(a))

given Monad[Identity]:
  def pure[A](a: A): Id[A] = Identity(a)
  def flatMap[A, B](f: A => Identity[B]): Identity[A] => Identity[B] =
    case Identity(a) => f(a)
```

可以看到 `Identity` 即是构造器/constructor，也是解构器/destructure，利用模式匹配是可以解构出值的。

上面函子实现中的 `fmap f (Identity a)`, 假如 `fmap` 的是 `Identity 1`, 那么这个模式匹配到 `(Identity a)` 时会通过解构器把 `1` 放到 `a` 的位置。

本来就有看起来什么也没有干，就跟 `identity` 函数一样，但是实际上, 它也跟 identity 相对于函数一样，在某些场景底下非常有用，比如后一部分搞基猫呢会提的高达猫。

15

Maybe 可能会有

可能会有单子/Maybe Monad 是一个超级简单的但比本身就有稍稍复杂的单子.

因为它拥有比本身就有多一个的类型构造器，类似这样的叫做代数数据类型/ Algebra Data Type(ADT)

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

其中 `a` [1]表示是任意类型.

你看, 不管是 `Just` 还是 `Nothing` 都可以构造出一个 `Maybe` 类型的数据来.

ADT 在 Scala 可以用 enum 表示, 而且, Scala 中的 `Maybe` 叫做 `Option`:

```scala
enum Option[+A]:
  case Some(a: A)
  case None
```

所以 `Just 1` 会得到一个 `Num a => Mabye a` 类型[2]，`Nothing` 也会得到一个 `Maybe a` 只不过 `a` 没有类型约束。

总之我们有了构造器可以构造出 `Maybe` 类型，而这个类型能做的事情，就要取决它实现了哪些 typeclass 的实例了。比如它可以是一个函子.

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

```scala
given Functor[Option]:
  def fmap[A, B](f: A => B): Option[A] => Option[B] =
    case Some(a) => Some(f(a))
    case None => None
```

看清楚了, 虚线箭头即 `fmap`, 图上表示的 `fmap` 是 `(a -> b) - - -> (Maybe a -> Maybe b)` 由于这里的箭头都是在 `->` 范畴, 所以 `- - ->` 就是 `->` 了.

即: `fmap :: (a -> b) -> f a -> f b`

[1] 一定要记得小写哦

[2] 意思就是 `Maybe a` 但是 `a` 的类型约束为 `Num`

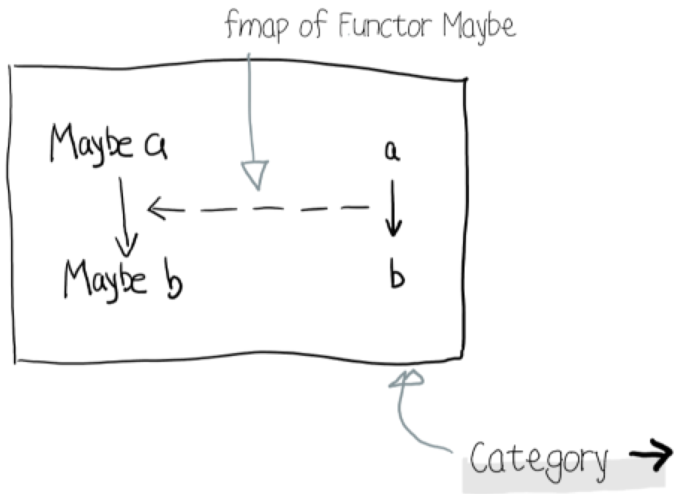


Figure 15.1: fmap :: (a -> b) -> f a -> f b

不仅如此，还可以实现单子:

```
instance Monad Maybe where
  return a = Just a
  (Just a) >>= f = f a
  Nothing >>= f = Nothing
```

```
given Monad[Option]:
  def pure[A](a: A): Option[A] = Some(a)
  def flatMap[A, B](f: A => Option[B]): Option[A] => Option[B] =
    case Some(a) => f(a)
    case None => None
  extension [A,B](fa: Option[A])
    def >>=(f: A => Option[B]): Option[B] = flatMap(f)(fa)
```

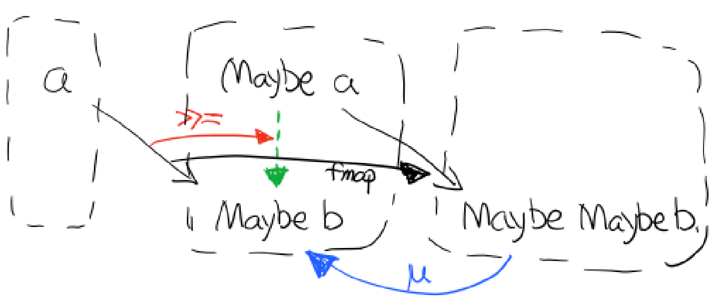


Figure 15.2: 还记得第一部分提到的 Kleisli 范畴吗?

Maybe 有用在于能合适的处理 偏函数 Partial Function/ 的返回值。偏函数相对于 全函数 Total Function/ 是指只能对部分输入返回输出的函数。

比如一个取数组某一位上的值的函数，就是偏函数，因为假设你想取第 4 位的值，但不是所有数组长度都大于 4，就会有获取不了的尴尬情况。

```
[1,2,3] !! 4
```

```
List(1,2,3).get(4)
```

如果使用 Maybe 把偏函数处理不了的输入都返回成 Nothing，这样结果依然保持 Maybe 类型，不影响后面的计算。

```
([[1,2,3], [4,5,6]] !! 1) >>= \x -> x !! 2
```

```
List(List(1,2,3), List(4,5,6)).get(1) >>= { _.get(2) }
```

16
Either 要么有要么有

Either 的定义也很简单

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

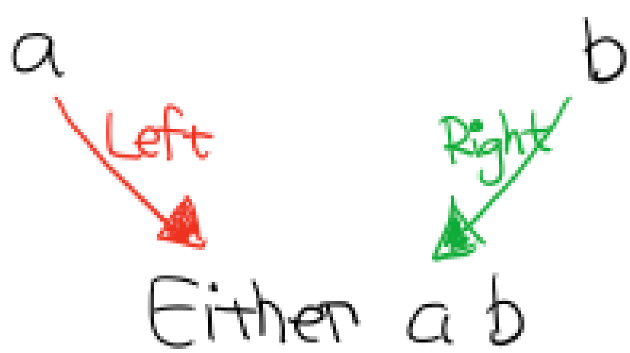
```
enum Either[+A, +B]:
  case Left(a: A)
  case Right(b: B)
```

16.1 Product & Coproduct

看过第一部分应该还能记得有一个东西叫 Duel，所以见到如果范畴上有 Coproduct 那么肯定在 duel 范畴上会有同样的东西叫 Product。

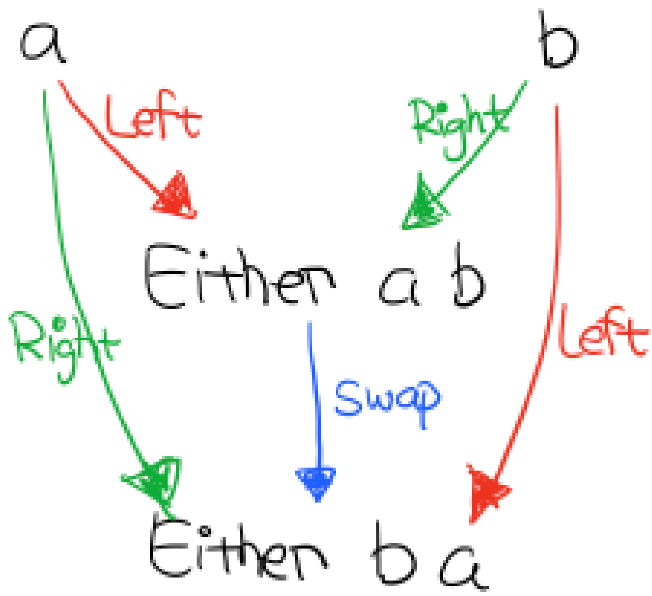
那么我们先来看看什么是 Coproduct

Figure 16.1: Coproduct



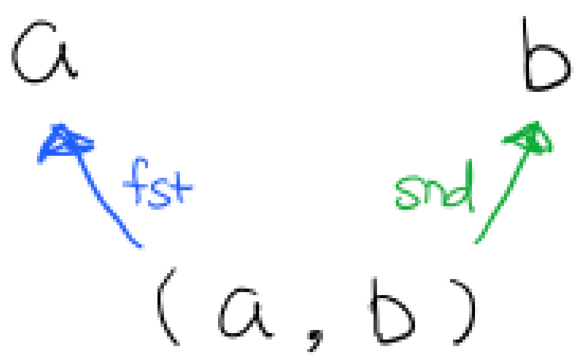
像这样，能通过两个箭头到达同一个东西，就是 Coproduct。这里箭头 `Left` 能让 `a` 到 `Either a b`，箭头 `Right` 也能让 `b` 到达 `Either a b`

有意思的是还肯定存在一个 Coproduct 和箭头，使得下图成立



箭头反过来，就是 Product，比如 Tuple

Figure 16.2: Product



Tuple 的 `fst` 箭头能让 `(a, b)` 到达 `a` 对象，而箭头 `snd` 能让其到达 `b` 对象。

16.2 Either Monad

确切的说，Either 不是 monad，`Either a` 才是。还记得 monad 的 class 定义吗？

```
class Endofunctor m => Monad m where
  eta :: a -> (m a)
  mu :: m m a -> m a
```

所以 m 必须是个 Endofunctor，也就是要满足 Functor

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

t a 的 kind 是 *，所以 t 必须是 kind * -> * 也就是说，m 必须是接收一个类型参数的类型构造器

而 Either 的 kind 是 * -> * -> *, Either a 才是 * -> *

所以只能定义 Either a 的 Monad

```haskell
instance Monad (Either a) where
  Left  l >>= _ = Left l
  Right r >>= k = k r
```

```scala
given [A]: Monad[Either[A, ?]] with
  def flatMap[B, C](f: B => Either[A, C]): Either[A, B] => Either[A, C] = (fa: Either[A, B]
    fa match
      case Left(l) => Left(l)
      case Right(r) => f(r)
```

很明显的，»= 任何函数到左边/ Left 都不会改变，只有 »= 右边才能产生新的计算。

17
Reader 差一点就有

差一点就有的作用是描述一个需要喂数据的计算。

在描述计算的时候，并不需要关心具体输入的值是什么，更需要关注的是输入的类型。当计算需要以来该值时，只需要 asks 就可以假装拿到输入值，继续描述接下来的计算。

而真正的输入，会在最终运行计算时给予。

跟 本身就有一样，我们用 newtype 来定义一个同构的 差一点就有类型：

```haskell
newtype Reader e a = Reader { runReader :: (e -> a) }
```

```scala
case class Reader[E, A](run: E => A)
```

其中：

- e 是输入
- a 是结果
- 构造 Reader 类型需要确定输入的类型 e 与输出的类型 a
- `runReader` 的类型是 `runReader:: (Reader e a) -> (e -> a)`

也就是说在描述完一个 Reader 的计算后，使用 runReader 可以得到一个 e -> a 的函数，使用这个函数，就可以接收输入，通过构造好的计算，算出结果 a 返回。

那么，让我们来实现 Reader 的单子实力，就可以描述一个可以 ask 的计算了。

```haskell
instance Monad (Reader e) where
    return a          = Reader $ \_ -> a
    (Reader g) >>= f = Reader $ \e -> runReader (f (g e)) e
```

```scala
given [E]: Monad[Reader[E, ?]] with
  def pure[A](a: A): Reader[E, A] = Reader((e: E) => a)
  def flatMap[A, B](f: A => Reader[E, B]): Reader[E, A] => Reader[E, B] = (fa: Reader[E, A]) =>
    Reader((e: E) => f(fa.run(e)).run(e))
```

跟 Either 一样，我们只能定义 Reader e 的 monad instance。
注意这里的

- f 类型是 `(a -> Reader e a)`
- g 其实就是是 destructure 出来的 runReader，也就是 e -> a
- 所以 (g e) 返回 a
- f (g e) 就是 `Reader e a`
- 再 run 一把最后得到 a

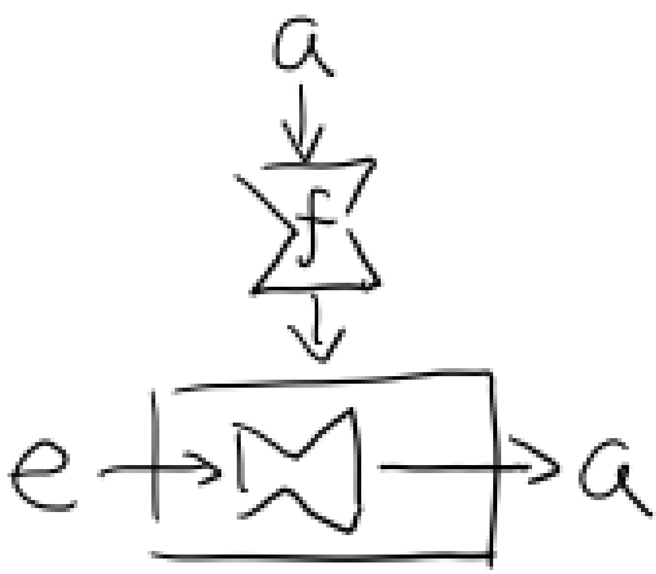


Figure 17.1: f 函数，接收 a 返回一个从 e 到 a 的 Reader

让我们来看看如何使用 Reader

```
import Control.Monad.Reader

data Environment = Env
  { fistName :: String
  , lastName :: String
  } deriving (Show)

helloworld :: Reader Environment String
helloworld = do
  f <- asks firstName
  l <- asks lastName
  return "Hello␣" ++ f ++ l
```

```
runHelloworld :: String
runHelloworld = runReader helloworld $ Env "Jichao" "Ouyang"
```

这段代码很简单，helloworld 负责打招呼，也就是在名字前面加个"Hello"，而跟谁打招呼，这个函数并不关心，而单纯的是向 Environment 问/asks 就好。

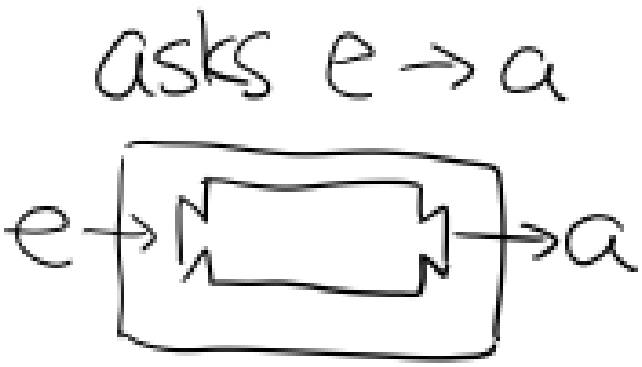
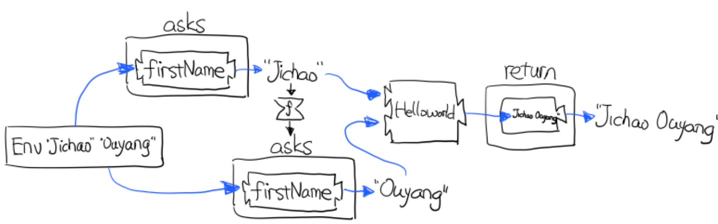


Figure 17.2: asks 可以将 e -> a 的函数变换成 Reader e a

在运行时，可以提供给 Reader 的输入 Env fistname lastname。



17.1 do notation

这可能是你第一次见到 `do` 和 `<-`. 如果不是，随意跳过这节。

- do 中所有 <- 的右边都是 `Reader Environment String` 类型
- do 中的 return 返回类型也必须为 `Reader Environment String`
- `asks firstName` 返回的是 `Reader Environment String` 类型，`<-` 可以理解成吧 monad `Reader Environment` 的内容放到左边的 f, 所以 f 的类型是 String。

看起来像命令式的语句，其实只是 `>>=` 的语法糖，但是明显用 do 可读性要高很多。

```
helloworld = (asks firstName) >>=
  \f -> (asks lastName) >>=
       \l -> return "Hello␣" ++ f ++ l
```

18 Writer 光出进没有

除了返回值，计算会需要产生一些额外的数据，比如 log

此时就需要一个 Writter，其返回值会是一个这样（`result`，`log`）的 tuple

限制是 log 的类型必须是个含幺半群/monoid

```
example :: Writer String String
example  = do
  tell "How are you?"
  tell "I'm fine thank you, and you?"
  return "Hehe Da~"

output :: (String, String)
output = runWriter example
-- ("Hehe Da~", "How are you?I'm fine thank you, and you?")
```

Writer 的定义更简单

```
newtype Writer l a = Writer { runWriter :: (a,l) }
```

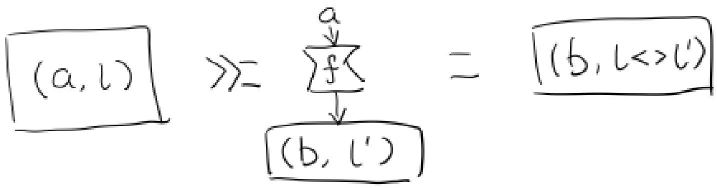
里面只是一个 tuple 而已

- w 是 log
- a 是返回值

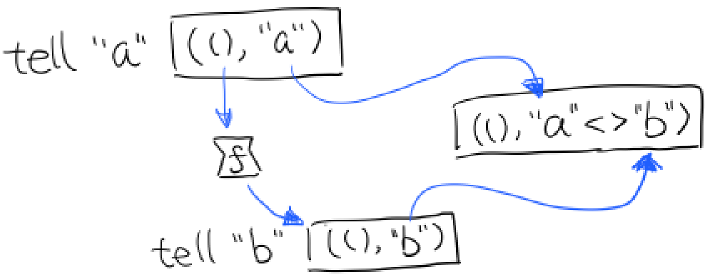
看看如何实现 Writer monad

```
instance (Monoid w) => Monad (Writer w) where
    return a             = Writer (a,mempty)
    (Writer (a,l)) >>= f = let (a',l') = runWriter $ f a in
                           Writer (a',l `mappend` l')
```

- return 不会有任何 log，l 是 monoid 的 mempty
- f 的类型为 `a -> Writer l a`
- `runWriter $ f a` 返回 `(a, l)`



所以在 »= 时，我们先把 f a 返回的 Writer run 了，然后把两次 log `mappend` 起来。



19

State 变化会有

跟名字就看得出来 State monad 是为了处理状态。虽然函数式编程不应该有状态，不然会引用透明性。但是，state monad 并不是在计算过程中修改状态，而是通过描述这种变化，然后需要时在运行返回最终结果。这一点跟 Reader 和 Writer 这两个看起来是副作用的 IO 是一样的。

先看下 State 类型的定义

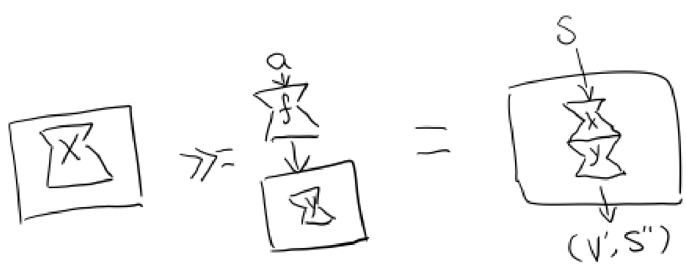
```
newtype State s a = State { runState :: s -> (a, s) }
```

可以看到 State 只包含一个从旧状态 s 到新状态 s 和返回值 a 的 Tuple 的函数。

通过实现 Monad，State 就可以实现命令式编程中的变量的功能。

```
instance Monad (State s) where
  return a        = State $ \s -> (a,s)
  (State x) >>= f = State $ \s -> let (v,s') = x s in
                                   runState (f v) s'
```

return 很简单，就不用解释了。



x 类型是 `s -> (a, s)`，所以 x s 之后会返回结果和状态。也就是运行当前 State，把结果 v 传给函数 f，返回的 State 再接着上次状态运行。

使用起来也很方便，State 提供 `get` `put` `moidfy` 三个方便的函数可以生成修改状态的 State monad

```
import Control.Monad.Trans.State.Strict
test :: State Int Int
```

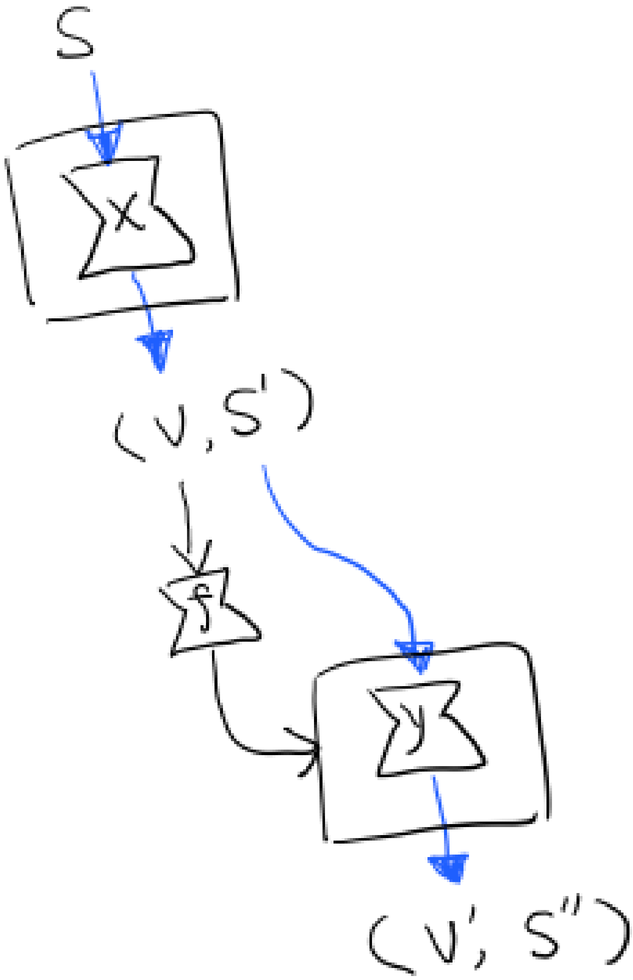


Figure 19.1: State x »= f 后 runState 的数据流（啊啊啊，画歪了，感觉需要脉动一下）

```
test = do
  a <- get
  modify (+1)
  b <- get
  return (a + b)

main = print $ show $ runState test 3
-- (7, 4)
```

20 *Validation* 检查检查

如果你有注意到，前面的 Either 可以用在处理错误和正确的路径分支，但是问题是错误只发生一次。

> Validation 没有在标准库中，但是我觉得好有用啊，你可以在 ekmett 的 github 中找到源码

想象一下这种场景，用户提交一个表单，我们需要对每一个 field 进行验证，如果有错误，需要把错误的哪几个 field 的错误消息返回。显然如果使用 Either 来做，只能返回第一个 field 的错误信息，后面的计算都会被跳过。

针对这种情况，Validation 更适合

```
data Validation e a = Failure e | Success a
```

ADT 定义看起来跟 Either 是一样的，不同的是左边/Left Failure 是含幺半群/Monoid

20.1 含幺半群/*Monoid*

monoid 首先得是半群/Semigroup ，然后再含幺。

```
class Semigroup a where
  (<>) :: a -> a -> a
  (<>) = mappend
```

半群非常简单，只要是可以 `<>` (mappend) 的类型就是了。

含幺只需要有一个 `mempty` 的幺元就行

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

比如 List 就是 Semigroup

```
instance Semigroup [a] where
  (<>) = (++)
```

也是 Monoid

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Monoid 的 <> 满足:

- mempty <> a = a
- a <> b <> c = a <> (b <> c)

20.2 回到 *Validation*

现在让 Failure e 满足 Monoid，就可以 mappend 错误信息了。

```haskell
instance Semigroup e => Semigroup (Validation e a) where
  Failure e1 <> Failure e2 = Failure (e1 <> e2)
  Failure _  <> Success a2 = Success a2
  Success a1 <> Failure _  = Success a1
  Success a1 <> Success _  = Success a1
```

下来，我们用一个简单的例子来看看 Validation 与 Either 有什么区别。

假设我们有一个 form，需要输入姓名与电话，验证需要姓名是非空而电话是 11 位数字。

首先，我们需要有一个函数去创建包含姓名和电话的 model

```haskell
data Info = Info {name: String, phone: String} deriving Show
```

然后我们需要验证函数

```haskell
notEmpty :: String -> String -> Validation [String] String
notEmpty desc "" = Failure [desc <> " cannot be empty!"]
notEmpty _ field = Success field
```

notEmpty 检查字符是否为空，如果是空返回 Failure 包含错误信息，若是非空则返回 Success 包含 field

同样的可以创建 11 位数字的验证函数

```haskell
phoneNumberLength :: String -> String -> Validation [String] String
phoneNumberLength desc field | (length field) == 11 = Success field
                             | otherwise = Failure [desc <> "'s length is not 11"]
```

实现 Validation 的 Applicative instance，这样就可以把函数调用 lift 成带有验证的 Applicative

```haskell
instance Semigroup e => Applicative (Validation e) where
  pure = Success
  Failure e1 <*> Failure e2 = Failure e1 <> Failure e2
  Failure e1 <*> Success _  = Failure e1
  Success _  <*> Failure e2 = Failure e2
  Success f <*> Success a = Success (f a)
```

- 失败应用到失败会 concat 起来
- 失败跟应用或被成功应用还是失败
- 只有成功应用到成功才能成功，这很符合验证的逻辑，一旦验证中发生任何错误，都应该返回失败。

```
createInfo :: String -> String -> Validation [String] Info
createInfo name phone = Info <$> notEmpty "name" name <*> phoneNumberLength "phone" phone
```

现在我们就可以使用带 validation 的 createInfo 来安全的创建 Info 了

```
createInfo "jichao" "12345678910" -- Success Info "jichao" "12345678910"
createInfo "" "123" -- Failure ["name cannot be empty!", "phone's length is not 11"]
```

21 *Cont* 接下来有

Cont 是 Continuation Passing Style/CPS 的 monad，也就是说，它是包含 cps 计算 monad。

先看一下什么是 CPS，比如有一个加法

```haskell
add :: Int -> Int -> Int
add = (+)
```

但是如果你想在算法加法后，能够继续进行一个其他的计算，那么就可以写一个 cps 版本的加法

```haskell
addCPS :: Int -> Int -> (Int -> r) -> r
addCPS a b k = k (a + b)
```

非常简单，现在我们可以看看为什么需要一个 Cont monad 来包住 CPS 计算，首先，来看 ADT 定义

```haskell
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
```

又是一个同构的类型，Cont 构造器只需要一个 runCount，也就是让他能继续计算的一个函数。

完了之后来把之前的 addCPS 改成 Cont

```haskell
add :: Int -> Int -> Cont k Int
add a b = return (a + b)
```

注意到 addCPS 接收到 a 和 b 之后返回的类型是 `(Int -> r) -> r`，而 Cont 版本的 `add` 返回 `Cont k Int`

明显构造 `Cont k Int` 也正是需要 `(Int -> r) -> r`，所以 Cont 就是算了 k 的抽象了。

```haskell
instance Monad (Cont r) where
    return a = Cont ($ a)
    m >>= k  = Cont $ \c -> runCont m $ \a -> runCont (k a) c
```

`($ a)` 比较有意思, 我们都知道 `f $ g a` 其实就是 `f(g a)`, 所以 `$` 其实就是一个 apply 左边的函数到右边表达式的中缀函数, 如果写成前缀则是 `($ (g a) f)`. 是反的是因为 `$` 是有结合, 需要右边表达式先求值, 所以只给一个 a 就相当于 `($ a) = \f -> f a`

回到 Monad Cont...

22
Summary

第二部分食用部分也讲完了，不知是否以及大致了解了 monad 的尿性各种基本玩法呢？通过这些常用的基本的 monad instance，解决命令式编程中的一些简单问题应该是够了。

不过，接下来还有更变态的猫，就先叫她 ~~搞基~~ 猫呢好了。

- 第三部分：搞基猫呢/ Advanced Monads

当然我又还没空全部写完，如果还有很多人预定/只要 998 Gumroad 上的电子书的话，我可能会稍微写得快一些。毕竟，写了也没人感兴趣也怪浪费时间的。不过，我猜也没几个人能看到这一行，就当是我又自言自语吧，怎么又突然觉得自己好分裂，诶 ~，为什么我要说又？

Part III

搞基猫呢/Advanced Monads

第二部分介绍了一些实用的 monad instances，这些 monad 都通过同样的抽象方式，解决了分离计算与副作用的工作。

通过它们可以解决大多数的基本问题，但是正对于复杂业务逻辑，我们可能还需要一些更高阶的 monad 或者 pattern。

当有了第一部分的理论基础和第二部分的实践，这部分要介绍的猫呢其实并不是很搞基。通过这一部分介绍的搞基猫呢，我们还可以像 IO monad 一样，通过 free 或者 Eff 自定义自己的计算，和可能带副作用的解释器。

23 RWS

RWS 是缩写 Reader Writer State monad, 所以明显是三个 monad 的合体。如果已经忘记 Reader Writer 或者 State，请到第二部分复习一下。

一旦把三个 monad 合体，意味着可以在同一个 monad 使用三个 monad 的方法，比如，可以同时使用 Reader 的 ask, State 的 get, put, 和 Writer 的 tell

```
readWriteState = do
  e <- ask
  a <- get
  let res = a + e
  put res
  tell [res]
  return res
runRWS readWriteState 1 2
-- (3 3 [3])
```

注意到跟 Reader 和 State 一样，run 的时候输入初始值

其中 1 为 Reader 的值，2 为 State 的初始状态.

24 Monad Transform

你会发现 RWS 一起用挺好的，能读能写能打 log，但是已经固定好搭配了，只能是 RWS ，如果我还想加入其它的 Monad，该怎么办呢?

这时候，简单的解决方案是加个 T，比如对于 Reader，我们有 ReaderT，RWS，也有对应的 RWST。其中 T 代表 Transform。

24.1 ReaderT

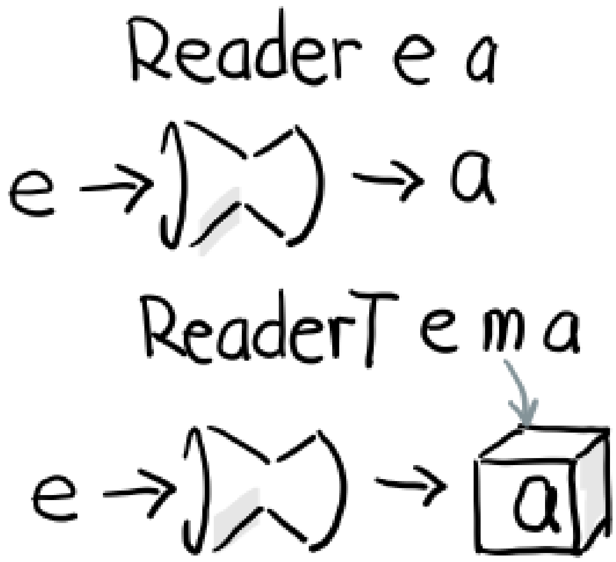
让我来通过简单的 ReaderT 来解释到底什么是 T 吧, 首先跟 Reader 一样我们有个 runReaderT

```
newtype ReaderT e m a = ReaderT { runReaderT :: e -> m a }
```

比较一下 Reader 的定义

```
newtype Reader e a = Reader { runReader :: (e -> a) }
```

有没有发现多了一个 m, 也就是说, `runReader e` 会返回 a, 但是 `runReaderT e` 则会返回 `m a`

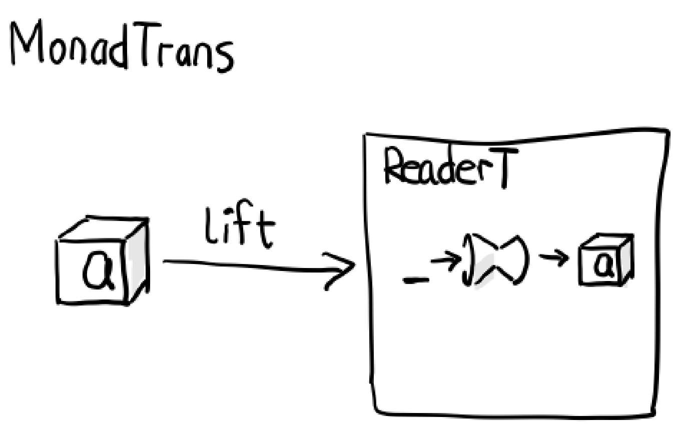



```
instance (Monad m) => Monad (ReaderT e m) where
    return   = lift . return
    r >>= k  = ReaderT $ \ e -> do
        a <- runReaderT r e
        runReaderT (k a) e
```

再看看 monad 的实现, 也是一样的, 先 run 一下 `r e` 得到结果 `a`, 应用函数 `k` 到 `a`, 再 run 一把.

问题是, 这里的 `return` 里面的 `lift` 是哪来的?

```
instance MonadTrans (ReaderT e) where
  lift m = ReaderT (const m)
```



这个函数 `lift` 被定义在 MonadTrans 的实例中, 简单的把 m 放到 ReaderT 结果中.

例如, `lift (Just 1)` 会得到 ReaderT, 其中 e 随意, m 为 Maybe Num

重点需要体会的是, Reader 可以越过 Maybe 直接操作到 Num, 完了再包回来.

有了 ReaderT, 搭配 Id Monad 就很容易创建出来 Reader Monad

```
type Reader r a= ReaderT r Identity a
```

越过 Id read 到 Id 内部, 完了再用 Id 包回来, 不就是 Reader 了么

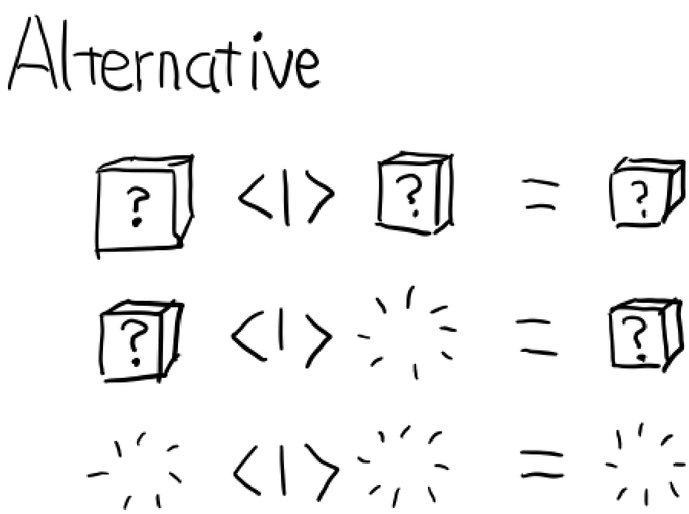
```
ReaderT { runReaderT :: r -> Identity a }
-- Identity a is a
ReaderT { runReaderT :: r -> a }
```

25

Alternative

这个 typeclass 提供 <|> 函数, 表示要么计算左边, 要么计算右边

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```



其实就是 Applicative 的 或
比如:

```
Just 1 <|> Just 2 -- Just 1
Just 1 <|> Nothing -- Just 1
Nothing <|> Just 1 -- Just 1
Nothing <|> Nothing -- Nothing
```

26
MonadPlus

这跟 Alternative 是一毛一样的, 只是限制的更细, 必须是 Monad 才行

```
class (Alternative m, Monad m) => MonadPlus m where
   mzero :: m a
   mzero = empty
   mplus :: m a -> m a -> m a
   mplus = (<|>)
```

看, 实现中直接就调用了 Alternative 的 `empty` 和 `<|>`

27
ST Monad

ST Monad 跟 State Monad 的功能有些像, 不过更厉害的是, 他不是 immutable 的, 而是"immutable" 的在原地做修改. 改完之后 runST 又然他回到了 immutable 的 Haskell 世界.

```
sumST :: Num a => [a] -> a
sumST xs = runST $ do           -- do 后面的事情会是不错的内存操作, runST 可以把它拉会纯的
    n <- newSTRef 0             -- 在内存中创建一块并指到 STRef
    forM_ xs $ \x -> do         -- 这跟命令式的for循环改写变量是一毛一样的
        modifySTRef n (+x)
    readSTRef n                 -- 返回改完之后的 n 的值
```

28 Free Monad

上一章说过的 RWS Monad 毕竟是固定搭配，当你的业务需要更多的 Monad 来表示 Effect 时，我们就需要有那么个小猪手帮我们定义自己的 Monad。

那就是 Free, Free 可以将任意 datatype lift 成为 Monad

28.1 Free

先看 Free 什么定义:

```
data Free f a = Roll (f (Free f a)) | Return a
```

其中 `f` 就是你业务需要的 effect 类型, `a` 是这个 effect 所产生的返回值类型。

右边两种构造函数，如果把 `Role` 改成 `Cons`, `Return` 改成 `Nil` 的话, 是不是跟 List 其实是同构/isomophic 的呢? 所以如果想象成 List, 那么 `f` 在这里就相当于 List 中的一个元素.

到那时, `>>=` 的操作又跟 List 略有不同, 我们都知道 `>>=` 会把每一个元素 map 成 List, 然后 flatten, 但 Free 其实是用来构建顺序的 effect 的, 所以:

```
instance Functor f => Monad (Free f) where
  return a        = Return a
  Return a >>= fn = fn a
  Roll ffa >>= fn = Roll $ fmap (>>= fn) ffa
```

你会发现 `>>=` 会递归的 `fmap` 到 `Roll` 上, 直到最后一个 `Return`.

比如, 如果你有一个 program 有三种副作用 Eff1, Eff2, Eff3

```
data Eff a = Eff1 a | Eff2 a | Eff3 a
program = do
 a <- liftF $ Eff1 1
 b <- liftF $ Eff2 2
 c <- liftF $ Eff3 3
 return a + b + c
```

如果我们把 program 展开, 每一步 `>>=` 大概是这样:

```
liftF $ Eff1 1
```

展开既是:

```
Roll (Eff1 (Return 1))
```

代入到 program 即:

```
program = Roll (Eff1 (Return 1)) >>= \a -> do
   b <- liftF $ Eff2 2
   c <- liftF $ Eff3 3
   return a + b + c
```

用 Free 的 `>>=` 公式 `Roll ffa >>= fn = Roll $ fmap (>>= fn) ffa` 去展开上面就得到:

```
program = Roll $ Eff1 (Return 1 >>= fn1)) where
  fn1 = \a -> do
   b <- liftF $ Eff2 2
   c <- liftF $ Eff3 3
   return a + b + c
```

`Return 1 >>= fn1` 我们都知道怎么展开:

```
program = Roll $ Eff1 (fn1 1) where
  fn1 = \a -> do
   b <- liftF $ Eff2 2
   c <- liftF $ Eff3 3
   return a + b + c
```

展开 `fn1`

```
program = Roll $ Eff1 do
   b <- liftF $ Eff2 2
   c <- liftF $ Eff3 3
   return 1 + b + c
```

同样的步骤展开 Eff2

```
program = Roll $ Eff1 $ Roll $ Eff2 do
   c <- liftF $ Eff3 3
   return 1 + 2 + c
```

和 Eff3

```
program = Roll $ Eff1 $ Roll $ Eff2 $ Roll $ Eff3 do
   return 1 + 2 + 3
```

最后的 program 是不是很像 List 的 Cons 和 Nil 呢?

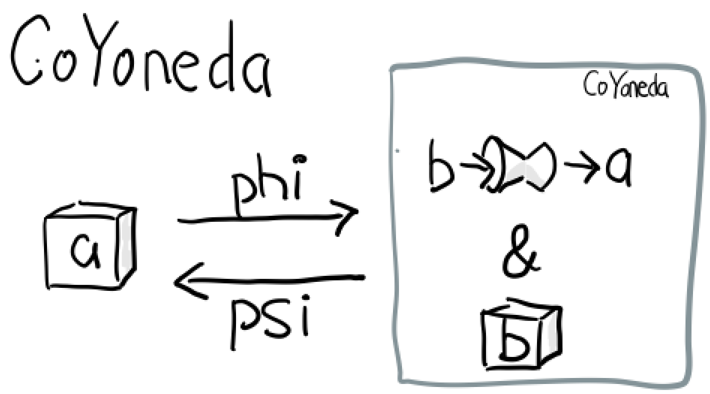
```
program = Roll $ Eff1 $ Roll $ Eff2 $ Roll $ Eff3 $ Return 1 + 2 + 3
```

但是, 细心的你可能早都发现了 `Eff` 这货必须是个 `Functor` 才行. 那我们如何随便定义一个 `data Eff` 直接能生成 `Functor Eff` 的实例呢?

28.2 Coyoneda

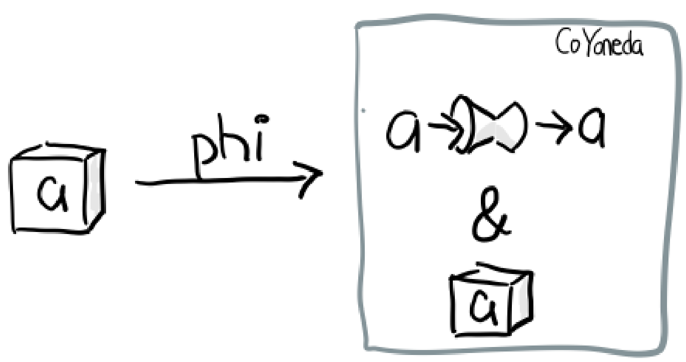
希望你还依然记得第一部分的米田 共 引理

```
data CoYoneda f a = forall b. CoYoneda (b -> a) (f b)
```



事实上很简单可以把任何 f 变成 CoYoneda f

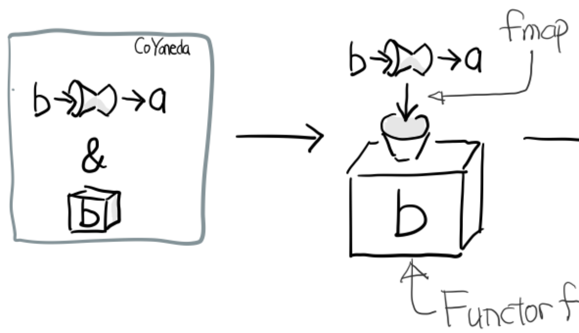
```
phi :: f a -> CoYoneda f a
phi fa = CoYoneda id fa
```



诀窍就是 id, 也就是你把 b 变成 a, 再把 fa 放到 CoYoneda 里就好了

当 f 是 Functor 时, 又可以把 CoYoneda 变成 f

```
psi :: Functor f => CoYoneda f a -> f a
psi (CoYoneda g fa) = fmap g fa
```



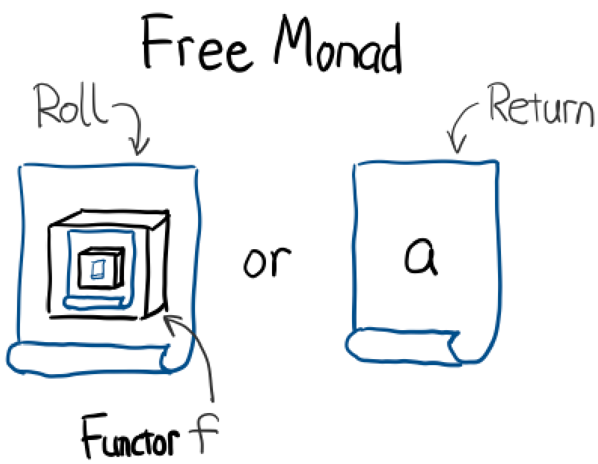
反过来的这个不重要, 重要的是 phi, 因为如果你可以把任何 f 变成 CoYoneda f, 而 CoYoneda f 又是 Functor, 我们不就免费得到一个 Functor?

```
instance Functor (Coyoneda f) where
  fmap f (Coyoneda g fb) = Coyoneda (f . g) fb
```

28.3 Free Functor

比如我们的 Eff 就可以直接通过 phi 变成 CoYoneda Eff, 从而得到免费的 Functor

```
data Eff a = Eff1 a | Eff2 a | Eff3 a
program = Roll (phi (Eff1 (Roll (phi (Eff2 (Return Int))))))
```



28.4 Interpreter

构造完一个 free program 后, 我们得到的是一个嵌套的数据结构, 当我们需要 run 这个 program 时, 我们需要 foldMap 一个 Interpreter 去一层层拨开这个 free program.

```haskell
foldMap :: Monad m => (forall x . f x -> m x) -> Free f a -> m a
foldMap _ (Return a)  = return a
foldMap f (Roll a) = f a >>= foldMap f
```

29

Free Monoid

30

Eff

31
Comonad