

文

<译> 函数类型 (/a/1190000004631638)

范晴论 (https://segmentfault.com/t/范晴论/blogs)

garfileo (https://segmentfault.com/u/garfileo)

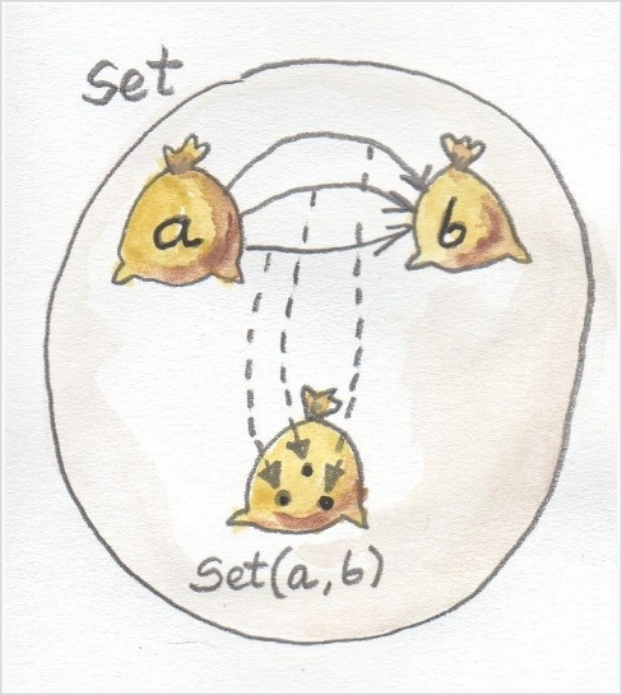
1 天前发布

原文见 <http://bartoszmilewski.com/2015/03/13/function-t...>

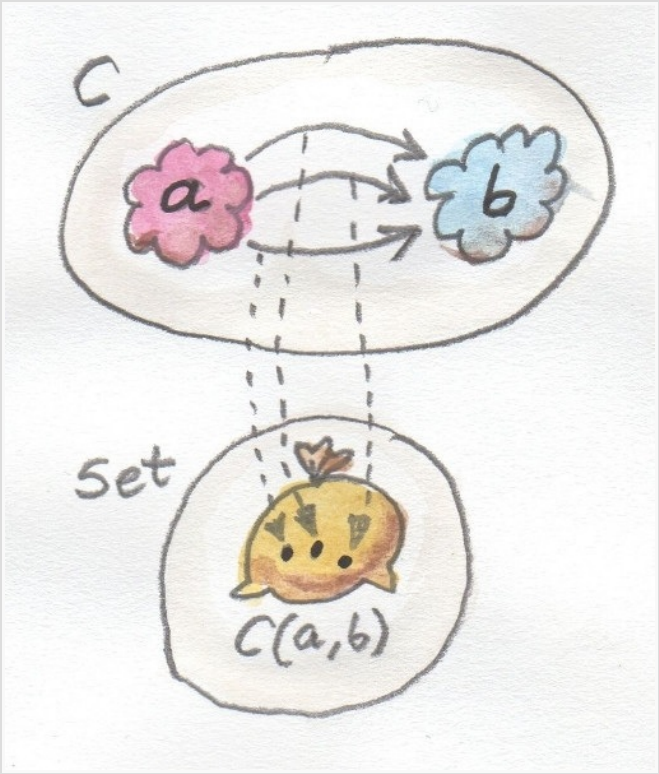
译注：由于距离上一章翻译的时间太久，所以有些内容可能有点不太相符。等我翻译完下一篇，再找时间校对一下。

到现在为止，我已多次语焉不详的提到函数类型。函数类型有别于其他类型。

以 `Integer` 为例，它是整数的集合。`Bool` 是两个元素构成的集合。然而一个函数类型 `a -> b` 的内涵要更大一些，它是从 `a` 到 `b` 的所有态射构成的集合。从一个对象到一个对象的态射所构成的集合，叫 `hom`-集。只有在集合的范畴中，`hom`-集自身也是一个对象——`hom`-集是一个集合。



对于其他范畴而言，`hom`-集会形成一个外部范畴。因而，这样的 `hom`-集被称为外 `hom`-集。



集合的范畴的自引用属性使得函数类型变得有些特殊。但是，至少在某些范畴中，有一种方法可以构造 `hom`-集这样的对象。这种 `hom`-集被称为内 `hom`-集。

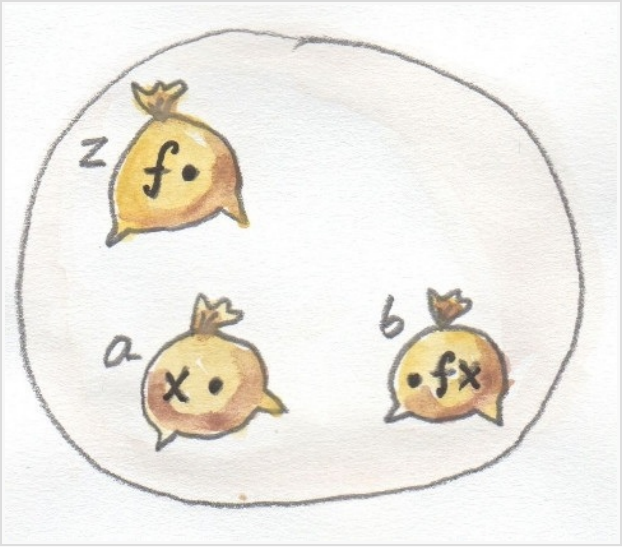
泛构造

暂时忘掉函数类型是集合这回事，我们先着手尝试去构造一个函数类型，或者从广义上说，去构造一个内 hom -集。像以前那样，我们可以从集合的范畴寻找线索，只需不触碰任何与集合有关的性质即可，这样所得到的结果对于其他范畴也是适用的。

可将函数类型视为一种复合类型，因为它描述的是一个参数类型（Argument Type）与结果类型（Result Type）之间的关系。之前我们见过这种复合类型的构造——那些包含了对象之间的关系的东西。我们用泛构造定义过积类型与余积类型 (<https://segmentfault.com/a/11...>)。可以使用同样的技巧来定义函数类型，前提是需给出可以联系三种对象的模式。这三种对象分别是：我们要构造的函数类型，参数类型及结果类型。

显然，联系这三种类型的模式就是**函数应用**（Function Application）或**求值**。对于函数类型，假设存在一个候选者 z （注意，在非集合的范畴中， z 只不过是个普通的对象），设参数类型为 a （一个对象），函数『应用』可将 z 与 a 构成的序对映射为结果类型 b （一个对象）。现在，我们有了三个对象了，它们中有两个是固定的（一个是参数类型，另一个是结果类型），剩下的那个就是『应用』。

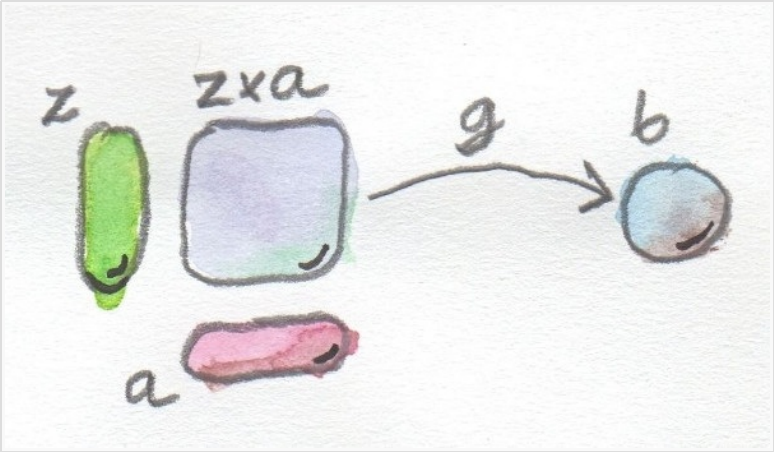
『应用』是一种映射，我们该如何将它融入到我们刚才所建立的模式之中？如果允许查看对象的内部，那么我们就可以将一个函数 f （ z 的一个元素）与参数 x 封装为序对，然后将这个序对映射为 $f \times x$ （ f 作用于 x ，所得结果就是 b 的一个元素）。



在集合的范畴中，可以从 z 这个函数集中拈取一个函数 f ，然后再从 a （类型）中拈取一个 x 作为参数，然后就可以得到 b （类型）中的一个元素 $f \times x$ 。

只是能处理单个的序对 (f, x) 是没有多少意思的，我们要处理的是函数类型的候选者 z 与参数类型 a 的积，即 $z \times a$ 。这个积是一个对象，可以选择从这个对象到 b 的一个箭头 g 作为态射， g 也就是『应用』。在集合的范畴中， g 就是将每个 (f, x) 映射为 $f \times x$ 的函数。

这样，我们就建立起了这样一个模式：对象 z 与对象 a 的积，通过态射 g 被关联到另一个对象 b 。

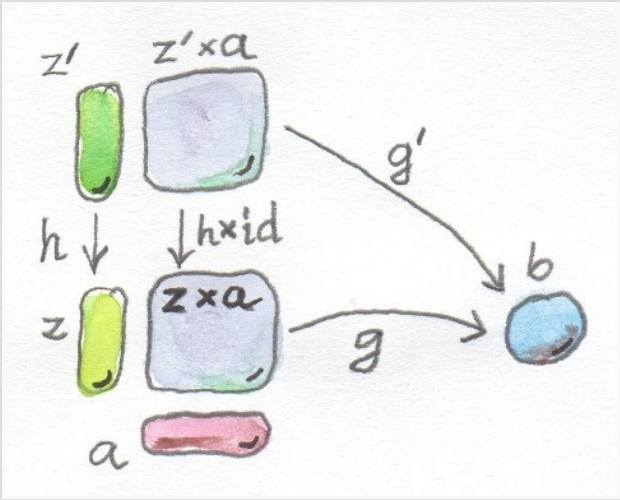


对象与态射构成的模式，它是函数类型的泛构造的起点。

要利用泛构造技巧来清晰的刻画函数类型，这样的模式够用么？这个模式不适于所有范畴，但是对于我们感兴趣的那些范畴，它够用了。还有一个问题：定义一个函数类型，必须事先定义积类型吗？有些范畴是不可积的，也就是对于其中的对象构成的序对不存在积类型。这个问题的答案是不行，如果没有积类型，就没有函数类型。等下文在讨论指数时再来谈这个问题。

现在回顾一下泛构造。我们以对象与态射构成的模式为起点，它是一种不精确的查询，通常会命中很多东西。特别是在集合的范畴中，几乎每一样东西都与其他东西具有相关性。我们可以拈取任意一个对象 z ，将其与 a 构成积，再找个函数将其映射为 b （除非 b 是空集）。

这样，我们的秘密武器——排名（Ranking）就派上了用场，就是检查一下是不是在函数类型的候选者之间存在唯一的映射——可以因子化泛构造的映射。当且仅当存在一个唯一的从 z' 到 z 的映射，使得 g' 可由 g 的因式来构造，即可判定伴随态射 g （从 $z \times a$ 到 b ）的 z 比伴随 g' 的候选者 z' 更好。（提示：请结合下图来阅读这段文字。）



在函数类型的候选者中建立排名机制

现在到了需要技巧的部分了，这也是我一直故意拖延着不谈这种特殊的泛构造的原因。假设存在态射 $h :: z' \rightarrow z$ ，我们想获得从 $z' \times a$ 到 $z \times a$ 的态射。由于积类型是具有函子性 (<https://segmentfault.com/a/11900000039...> 的，因此就知道该怎么做了。由于积类型本身是一个函子（更确切的说，是二元自函子），因此可以提升一对态射。也就是说，我们不仅能定义对象的积，也能定义态射的积。

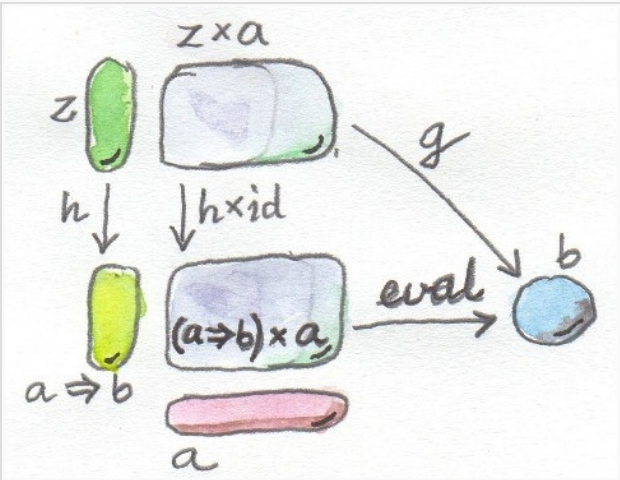
由于不需要改变 $z' \times a$ 这个积的第二个成员 a ，因此我们要提升的态射序对是 (h, id) ，其中 id 是作用于 a 的恒等态射。

现在我们可以建立『应用』的因式了，将 g' 表示为 g 的因式：

$$g' = g \circ (h \times id)$$

这里，关键之处在于态射积的作用（译注：它扮演了一个系数的角色）。

泛构造的第三步就是选出最好的那个候选者——我们称之为 $a \Rightarrow b$ （在这里，将它视为一个对象的名字即可，不要与 Haskell 类型类的约束符号混淆，下文会给出其他命名方式）。这个对象伴随的『应用』——从 $(a \Rightarrow b) \times a$ 到 b 的态射——我们称之为 $eval$ 。如果其他候选者所对应的『应用』 g 都能由 $eval$ 的『因式』被唯一的构造出来，那么 $a \Rightarrow b$ 就是最好的那个候选者。



泛函数对象的定义。此图与上面那副图相同，但是现在 $a \Rightarrow b$ 是『泛』的。

形式定义如下：

一个从 a 到 b 的函数对象是伴随着态射

$$eval :: ((a \Rightarrow b) \times a) \rightarrow b$$

的 $a \Rightarrow b$ ，它对于伴随着态射

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

的任意其他对象 z 而言，存在唯一的态射

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

g 可表示为 h 与 $eval$ 所构成的因式：

```
g = eval ∘ (h × id)
```

虽然不能担保对于某个范畴中的任意的对象 a 与 b 都存在着 $a \Rightarrow b$ ，但是对于集合的范畴却总是存在着这样的 $a \Rightarrow b$ ，并且在集合的范畴中， $a \Rightarrow b$ 与 Hom-集 **Set**(**a**,**b**) 同构。这也是为何在 Haskell 中我们将函数类型 $a \rightarrow b$ 解释为范畴意义上的函数对象 $a \Rightarrow b$ 的原因。

柯里化

现在再次观察函数类型的所有候选者。不过，这次我们将态射 g 视为两个变量 z 与 a 的函数：

```
g :: z × a -> b
```

一个接受积类型的态射与一个接受两个变量的函数很相似。特别是在集合的范畴中， g 是接受值的序对的函数，其中一个值来自集合 z ，另一个来自集合 a 。

另一方面，泛性质告诉我们，对于每个这样的 g ，都存在唯一的态射 h ——将 z 映射为一个函数类型 $a \Rightarrow b$ ：

```
h :: z -> (a⇒b)
```

在集合的范畴中，这恰恰意味着 h 是一个接受 z 类型并返回一个从 a 到 b 的函数的函数。也就是说， h 是一个高阶函数。因此这种泛构造在「接受两个变量的函数」与「接受一个变量并返回函数的函数」之间建立了壹壹对应的关系。这种对应关系叫做**柯里化**，并且 h 叫做 g 的柯里化版本。

这种对应关系一对一的，因为对于任意 g 都存在唯一的 h ，而对于任意的 h ，总是能重建一个接受 2 个参数的函数 g ，即：

```
g = eval ∘ (h × id)
```

可将函数 g 称为 h 的**反柯里化**版本。

柯里化是 Haskell 内建的语法。返回一个函数的函数：

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

经常被视为一个接受两个参数的函数，也就是将上述的函数签名中的括号去掉，即：

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

接受多个参数的函数，这种解释比较符合我们的直觉。例如：

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

这个函数也可以表示为下面的形式——接受一个参数然后返回一个函数（匿名函数）的函数：

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

这两种定义是等价的，并且二者都可以部分应用于 1 个参数，结果产生一个单参数的函数，例如：

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

严格的说，接受 2 个参数的函数，它本质上接受的是一个序对（积类型）：

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

这两种形式的相互转换有点微不足道，有两个函数可以实现它们的相互转换。没什么可奇怪的，这两个函数分别叫 `curry` 与 `uncurry`：

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

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

注意，`curry` 是函数类型泛构造的因子生成器，将其写为下面的形式会更清晰一些（译注：我却越来越觉得混乱了！）：

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

因式生成器基于候选者生成因子 `eval`。

在非函数式语言中，例如 C++，可以实现柯里化，但不是那么简单。可将 C++ 中的多参数函数视为接受元组（Tuple）的 Haskell 函数（尽管有些东西相当混乱，在 C++ 中可以定义显式接受 `std::tuple` 的函数，也可以定义变参函数，还可以定义接受已初始化的列表的函数）。

借助模板 `std::bind`，可实现 C++ 函数的部分应用。例如下面接受两个字符串的函数：

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

可基于这个函数定义接受一个字符串的函数：

```
using namespace std::placeholders;

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

虽然 Scala 要比 C++ 或 Java 更加的函数化，但是它却做不到函数的部分应用。如果你预期自己所定义的函数将会被部分应用，不得不借助多参数列表：

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

库的作者对于这部分函数的定义需要具有前瞻性。

指数

在数学领域，从对象 `a` 到对象 `b` 的函数或内 hom-对象（hom-集合中的对象）通常称为指数，表示为 `ba`。注意，函数参数类型位于指数的位置。这种形式看上去会很奇怪，但是当你了解了函数与积的关系时，就能领会这种形式的美妙之处。我们已经见识了必须借助积来实现内 hom-对象的泛构造，然而函数与积还有更深的联系。

考虑一下那些建立在有限类型——值的数量为有限的类型，例如 `Bool`、`Char`，甚至 `Int` 或 `Double` 之上的函数。至少在理论上，这些函数可被记忆化，亦即可将这些函数转化为表，然后通过查表的方式获得函数返回值。这是函数（态射）与函数类型（对象）之间等价的本质。

例如，一个接受 `Bool` 的（纯）函数可以被特化为一对值：一个对应于 `False`，另一个对应于 `True`。比方说，所有从 `Bool` 到 `Int` 的函数等价于所有 `Int` 序对所构成的集合，用积来表示就是 `Int × Int`，或者再有点创造性，可将其表示为 `Int2`。

再看一个例子，C++ 的 `char` 类型，它包含 256 个值。在 C++ 标准库中经常会使用数据查询的方式来定义一些函数。例如 `isupper` 或 `isspace` 都是用表来实现的，它等价于 256 个布尔值构成的元组。元组是积类型，因此我们要处理的是 256 个 `Bool` 值的积：`bool × bool × bool × ... × bool`。在算术中，重复的积就是幂。如果你将 `bool` 乘以它自身 256（或 `char`）次，那么你得到的就是 `bool{char}`。

`bool` 的 256-元组一共有多少个？答案是 2²⁵⁶ 个。这也就是从 `char` 到 `bool` 的不同函数的总数，每个函数对应唯一的 256-元组。同理，从 `bool` 到 `char` 的函数数量是 256²。在这些例子中，函数类型的指数表示相当美妙。

我们可能并不想将一个接受 `int` 或 `double` 的函数完全的记忆化，因为这样不切实际，但是函数与数据类型之间的等价性总是客观存在的。还有一些有限类型，例如列表、字符串或树，如果将接受这些类型的函数进行记忆化，需要无限的存储空间。然而 Haskell 是一种惰性语言，因此在惰性求值的（无限的）数据结构与函数之间的界限并不那么明显。这种函数与数据之间的对偶性揭示了 Haskell 的函数类型与范畴化的指数对象之间的等价性——我们又朝向数据迈进了一步。

笛卡尔闭范畴

尽管我会继续使用集合范畴作为类型与函数的模型，但是要注意很大一部分范畴也能够胜任这项任务。这些范畴被称为笛卡尔闭的，集合范畴就属于此类范畴。

一个笛卡尔闭范畴必须包含：

1. 终端对象
2. 任意对象序对的积
3. 任意对象序对的指数

如果你能将指数想象为重复的积（可能是无限次），那么你就可以将笛卡尔闭范畴想象为支持任意数量的积运算的范畴。特别是，可将终端对象想象为 0 个对象的积，或者一个对象的 0 次幂。

从计算机科学的角度来看，笛卡尔闭范畴的有趣之处在于它为简单的类型 Lambda 演算——所有类型化的编程语言的基础—提供了模型。

终端对象与积也分别具有对偶物：初始对象与余积。笛卡尔闭范畴也支持后者，积通过分配率可转化为余积：

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

这样的范畴被称为双向笛卡尔闭范畴。在下一节中就会看到集合范畴是这种范畴的基本范例以及这种范畴一些有趣的性质。

指数与代数数据类型

从指数的角度来阐释函数类型，这种方式也能很好的适用于代数数据类型。事实上，中学代数中所涉及的 0，1，加法，乘法以及指数等结构，在任何双向笛卡尔闭范畴中同样存在，它们分别对应于初始对象，终端对象，余积，积以及指数等。我们现在还没有足够的工具（诸如伴随（Adjunction）或 Yoneda 定理）来证明这一点，不过在此我可以将此直观的呈现出来。

0 次幂

$$a^0 = 1$$

在范畴论中，0 即初始对象，1 即终端对象，『相等』即恒等态射。指数即内 hom-对象。上面这个特殊的指数表示的是从初始对象到任意对象 a 的态射集合。基于初始对象的定义，这样的态射只有一个，因此 hom-集 C(0, a) 是一个单例集合。一个单例集合在集合范畴中是终端对象，因此上面这个等式在集合范畴中是成立的，这也意味着它在任何双向笛卡尔闭范畴中都成立。

在 Haskell 中，我们用 `Void` 表示 0，用 `unit` 类型 `()` 表示 1，用函数类型表示指数。所有从 `Void` 到任意类型 `a` 的函数集合等价于 `unit` 类型——单例集合。换句话说，有且仅有一个函数 `Void -> a`，这个函数之前我们已经见识了，它叫 `absurd`。不过，这多少有点投机倒把，原因有二。第一，在 Haskell 不存在没有值的类型——每种类型都包含着『永不休止的运算』，即底。第二，`absurd` 的所有实现都是等价的，因为无论用那种方式实现它们，也没人能够执行它们——没有值可以传递给 `absurd`。（如果你传递给它一个永不休止的运算，它什么也不会返回！）

1 的幂

$$1^a = 1$$

在集合范畴中，这个等式重申了终端对象的定义：从任意对象到终端对象存在唯一的态射。从 `a` 到终端对象的内 home-对象通常与终端对象本身是同构的。

在 Haskell 中，只有一个函数是从任意类型 `a` 到 `unit` 类型的，之前见过这个函数——`unit`。你可以认为它是 `const` 函数对 `()` 的偏应用。

1 次幂

$$a^1 = a$$

这个等式重申了从终端对象出发的态射可用于从对象 `a` 中拮取元素。这种态射的集合与对象 `a` 本身是同构的。在集合范畴与 Haskell 中，集合 `a` 与从 `a` 中拮取元素的函数 `() -> a` 是同构的。

指数的和

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

从范畴论的角度来看，这个等式描述的幂为两个对象的余积的指数与两个指数的积同构。在 Haskell 中，这种代数等式有着非常特别的解释，它告诉了我们，两个类型的和的函数与两个参数类型为单一类型的函数的积同构。这恰恰就是我们在定义作用于和类型的函数时所用到的分支分析，也就是说，函数定义中的 `case` 语句可以用两个或多个处理特定类型的函数来替代。例如，下面这个从和类型 `(Eigher Int Double)` 出发的函数 `f`：

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

可以定义为一个函数对：

```
f (Left n)  = if n < 0 then "Negative int"  else "Positive int"
f (Right x) = if x < 0.0 then "Negative double" else "Positive double"
```

在此，`n` 是 `Int`，而 `x` 是 `Double`。

指数的指数

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

这个等式表达的是指数对象形式的柯里化——返回一个函数的函数等价与积类型的函数（带两个参数的函数）。

积的指数

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

在 Haskell 中，返回一个序对的函数与一对函数等价，后者的每个函数都返回序对的一个元素。

这些中学数学里的等式可以提升至范畴论中并且在函数式编程中获得应用，这一切非常不可思议。

柯里-霍华德同构

我曾提到过逻辑学与代数数据类型之间的一些对应。`Void` 类型与 `unit` 类型 `()` 分别对应于错误与正确。积类型与和类型分别对应于逻辑与运算 `\wedge` 与逻辑或运算 `\vee`。遵循这一模式，我们所定义的函数类型对应于逻辑推理 `\Rightarrow`，换句话说，类型 `a -> b` 可以读为『如果 `a` 那么 `b`』。

根据柯里-霍华德同构理论，每种类型可视为一个命题——为真或为假的陈述语句。如果类型是有值的，那么它就是真命题，否则就是伪命题。在实践中，如果一个函数类型有值，亦即存在这样的函数，那么与它对应的逻辑推理就为真。去实现一个函数，就是在证明一个定理。写程序，就等价于证明许多定理。下面看几个例子。

以函数类型定义中所用的 `eval` 函数为例，它的签名是：

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

它接受一个由函数与其参数构成的序对，产生相应的类型。这个函数是一个态射的 Haskell 实现，该态射为：

```
eval :: (a=>b) × a -> b
```

这个态射定义了函数类型 $a \Rightarrow b$ （或指数类型 b^a ）。运用柯里-霍华德同构理论，可将这个签名转化为逻辑命题：

$$((a \rightarrow b) \wedge a) \rightarrow b$$

可将上面这条陈述读为：如果 b 由 a 推出为真，并且 a 为真，那么 b 肯定为真。这就是所谓的肯定前件式。要证明这个定理，只需要实现一个函数，即：

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

如果你给我一个从 a 到 b 的函数 f 以及 a 类型的一个值 x 所构成的序对，我就可以将 f 作用于 x ，从而产生 b 类型的一个值。通过实现这个函数，我可以证明 $((a \rightarrow b), a) \rightarrow b$ 是有值的。因此，在我们的逻辑中，这一肯定前件式为真。

结果为假的逻辑命题是怎样的？看这个例子，如果 a 或 b 为真，那么 a 肯定为真：

$$a \vee b \rightarrow a$$

这个命题肯定是错的，因为当 a 为假而 b 为真时，就可以构成一个反例。运用柯里-霍华德同构理论，可将这个命题映射为函数签名：

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

你可以试试看，根本无法实现这样的函数，因为对于 `Right` 构造的值而言，无法产生类型为 a 的值。（注意，我说的是纯函数）。

最后，来理解一下 `absurd` 函数：

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

将 `Void` 视为假，可得：

$$\text{false} \rightarrow a$$

这意味着由谎言可推理出一切（爆炸原理）。对于这个命题（函数），下面用 Haskell 给出的一个证据（实现）：

```
absurd (Void a) = absurd a
```

其中 `Void` 的定义如下：

```
newtype Void = Void Void
```

这是我们惯用的花招，这个定义使得 `Void` 不可能用于构造一个值，因为你要用它构造一个值，前提是必须先提供这个类型的一个值。这样就可以使得 `absurd` 永远无法被调用。

这些例子很有趣，但是在现实中能够用柯里-霍华德同构吗？平时可能用不到，但是像 Agda 或 Coq 这样的编程语言，它们可以利用柯里-霍华德同构来证明定理。

计算机不仅仅是数学家的辅助工具，它正在掀起基础数学的一场革命。在这个领域，最新的研究热点是同伦类型论（Homotopy type theory），它是类型理论的庞枝，也涉及布尔类型、整型、积、余积以及函数类型等结构，并且似乎是要驱散人们对它是否有用的质疑，Coq 与 Agda 将这一理论进行了形式化构建。计算机对这个世界的革命途径不止一种。

参考文献

Ralph Hinze, Daniel W. H. James, Reason Isomorphically!. 这份论文给出了本章所提到的中学代数等式在范畴论中的对应结构的证明。

致谢

感谢 Gershom Bazerman 检查了本章在数学和逻辑方面的内容。感谢 André van Meulebrouck 对本系列文章内容编排上的帮助。

1 天前发布 (/a/1190000004631638)

你可能感兴趣的文章

- <译> 写给程序猿的范畴论 · 序 (https://segmentfault.com/a/1190000003882331) 34 收藏，2.1k 浏览
- <译> 范畴，可大可小 (https://segmentfault.com/a/1190000003894116) 4 收藏，1k 浏览
- <译> 范畴：复合的本质 (https://segmentfault.com/a/1190000003883257) 10 收藏，1.6k 浏览

讨论区

好久不见你翻范畴的东西了

(https://segmentfault.com/c/1050000004632977) 吴隐隐 (https://segmentfault.com/u/wuyinyin) · 1 天前

请先 登录 () 后评论




(https://sponsor.segmentfault.com/ck.php?oaparams=2_bannerid=7_zoneid=2_cb=bc8bbeaef9_oadest=http://click.aliyun.com/m/3936/)

本文隶属于专栏

cd / (https://segmentfault.com/blog/root)

回到系统根目录.....

 garfileo (https://segmentfault.com/u/garfileo)
作者

关注专栏

分享扩散：

...