Currying 的局限性

很多基于 lambda calculus 的程序语言，比如 ML 和 Haskell，都习惯用一种叫做 currying 的手法来表示函数。比如，如果你在 Haskell 里面这样写一个函数：

```
f x y = x + y
```

然后你就可以这样把链表里的每个元素加上 2：

```
map (f 2) [1, 2, 3]
```

它会输出 `[3, 4, 5]`。

注意本来 `f` 需要两个参数才能算出结果，可是这里的 `(f 2)` 只给了 `f` 一个参数。这是因为 Haskell 的函数定义的缺省方式是“currying”。Currying 其实就是用“单参数”的函数，来模拟多参数的函数。比如，上面的 `f` 的定义在 Scheme 里面相当于：

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

它是说，函数 `f`，接受一个参数 `x`，返回另一个函数（没有名字）。这个匿名函数，如果再接受一个参数 `y`，就会返回 `x + y`。所以上面的例子里面，`(f 2)` 返回的是一个匿名函数，它会把 2 加到自己的参数上面返回。所以把它 map 到 `[1, 2, 3]`，我们就得到了 `[3, 4, 5]`。

在这个例子里面，currying 貌似一个挺有用的东西，它让程序变得“简短”。如果不用 currying，你就需要制造另一个函数，写成这个样子：

```
map (\y->f 2 y) [1, 2, 3]
```

这就是为什么 Haskell 和 ML 的程序员那么喜欢 currying。这个做法其实来源于最早的 lambda calculus 的设计。因为 lambda calculus 的函数都只有一个参数，所以为了能够表示多参数的函数，有一个叫 Haskell Curry 的数学家和逻辑学家，发明了这个方法。

当然，Haskell Curry 是我很尊敬的人。不过我今天想指出的是，currying 在程序设计的实践中，其实并不是想象中的那么好。大量使用 currying，其实会带来程序难以理解，复杂性增加，并且还可能因此引起意想不到的错误。

不用 currying 的写法`(\y->f 2 y)`虽然比起 currying 的写法`(f 2)`长了那么一点，但是它有一点好。那就是你作为一个人（而不是机器），可以很清楚的从“\y->f 2 y”这个表达式，看到它的“用意”是什么。你会很清楚的看到：

> “f 本来是一个需要两个参数的函数。我们只给了它第一个参数 2。我们想要把 [1, 2, 3] 这个链表里的每一个元素，放进 f 的第二个参数 y，然后把 f 返回的结果一个一个的放进返回值的链表里。”

仔细看看上面这段话说了什么吧，再来看看 `(f 2)` 是否表达了同样的意思？注意，我们现在的“重点”在于你，一个人，而不在于计算机。你仔细想，不要让思维的定势来影响你的判断。

你发现了吗？`(f 2)` 并不完全的含有 `\y->f 2 y` 所表达的内容。因为单从 `(f 2)` 这个表达式（不看它的定义），你看不到“f 总共需要几个参数”这一信息，你也看不到 (f 2) 会返回什么东西。f 有可能需要2个参数，也有可能需要3个，4个，5个…… 比如，如果它需要3个参数的话，`map (f 2) [1, 2, 3]` 就不会返回一个整数的链表，而会返回一个函数的链表，它看起来是这样：`[(\z->f 2 1 z), (\z->f 2 2 z), (\z->f 2 3 z)]`。这三个函数分别还需要一个参数，才会输出结果。

这样一来，表达式 `(f 2)` 含有的对“人”有用的信息，就比较少了。你不能很可靠地知道这个函数接受了一个参数之后会变成什么样子。当然，你可以去看 `f` 的定义，然后再回来，但是这里有一种“直觉”上的开销。如果你不能同时看见这些信息，你的脑子就需要多转一道弯，你就会缺少一些重要的直觉。这种直觉能帮助你写出更好的程序。

然而，currying 的问题不止在于这种“认知”的方面，有时候使用 curry 会直接带来代码复杂性的增加。比如，如果你的 f 定义不是加法，而是除法：

```
f x y = x / y
```

然后，我们现在需要把链表 `[1, 2, 3]` 里的每一个数都除以 2。你会怎么做呢？

`map (f 2) [1, 2, 3]` 肯定不行，因为 2 是除数，而不是被除数。熟悉 Haskell 的人都知道，可以这样做：

```
map (flip f 2) [1, 2, 3]
```

`flip` 的作用是“交换”两个参数的位置。它可以被定义为：

```
flip f x y = f y x
```

但是，如果 f 有 3 个参数，而我们需要把它的第 2 个参数 `map` 到一个链表，怎么办呢？比如，如果 `f` 被定义为：

```
f x y z = (x - y) / z
```

稍微动一下脑筋，你可能会想出这样的代码：

```
map (flip (f 1) 2) [1, 2, 3]
```

能想出这段代码说明你挺聪明，可是如果你这样写代码，那就是缺乏一些“智慧”。有时候，好的程序其实不在于显示你有多“聪明”，而在于显示你有多“笨”。现在我们就来看看笨一点的代码：

```
map (\y -> f 1 y 2) [1, 2, 3]
```

现在比较一下，你仍然觉得之前那段代码很聪明吗？如果你注意观察，就会发现 `(flip (f 1) 2)` 这个表达式，是多么的晦涩，多么的复杂。

从 `(flip (f 1) 2)` 里面，你几乎看不到自己想要干什么。而 `\y-> f 1 y 2` 却很明确的显示出，你想用 1 和 2 填充掉 `f` 的第一，三号参数，把第二个参数留下来，然后把得到的函数 map 到链表 `[1, 2, 3]`。仔细看看，是不是这样的？

所以你花费了挺多的脑力才把那使用 currying 的代码写出来，然后你每次看到它，还需要耗费同样多的脑力，才能明白你当时写它来干嘛。你是不是吃饱了没事干呢？

练习题：如果你还不相信，就请你用 currying 的方法（加上 `flip`）表达下面这个语句，也就是把 f 的第一个参数 map 到链表 `[1, 2, 3]`：

```
map (\y -> f y 1 2) [1, 2, 3]
```

得到结果之后再跟上面这个语句对比，看谁更加简单？

到现在你也许注意到了，以上的“笨办法”对于我们想要 `map` 的每一个参数，都是差不多的形式；而使用 currying 的代码，对于每个参数，形式有很大的差别。所以我们的“笨办法”其实才是以不变应万变的良策。

才三个参数，currying 就显示出了它的弱点，如果超过三个参数，那就更麻烦了。所以很多人为了写 currying 的函数，特意把参数调整到方便 currying 的顺序。可是程序的设计总是有意想不到的变化。有时候你需要增加一个参数，有时候你又想减少一个参数，有时候你又会有别的用法，导致你需要调整参数的顺序…… 事先安排好的那些参数顺序，很有可能不能满足你后来的需要。即使它能满足你后来的需要，你的函数也会因为 currying 而难以看懂。

这就是为什么我从来不在我的 ML 和 Haskell 程序里使用 currying 的原因。古老而美丽的理论，也许能够给我带来思想的启迪，可是未必就能带来工程中理想的效果。