函数式语言的宗教

很早的时候，“函数式语言”对于我来说就是 Lisp，因为 Lisp 可以在程序的几乎任意位置定义函数，并且把它们作为值来传递（这叫做 first-class function）。可是到后来有人告诉我，Lisp 其实不算“函数式语言”，因为 Lisp 的函数不“纯”（pure）。

所谓“纯函数”，就是像数学函数一样，你给它同样的输入，它就给你同样的输出。然后你就发现在这种定义下，几乎所有程序语言里面的随机数函数（`random`），都不是“纯函数”。因为每一次调用 `random()`，输入都是一样的（没有参数），但每次会输出不同的随机数。他们告诉我，不纯的函数容易出错，没法验证它的正确性。

在这种害怕自己用的语言“不纯”，不安全的恐慌之下，我开始接触 Haskell，一种号称“纯函数式”，安全的语言。我深信 Haskell 的纯函数教条长达几年之久，对其它语言里的“副作用”（side-effect）嗤之以鼻。我认为宇宙的本质是纯的，数学就是宇宙的终极语言，所以程序语言也应该像数学一样纯粹……

你有没有意识到，所有的邪教头子最初都是利用了人们的恐惧心理，进而让他们深信不疑，以为遇到了救世主的？当我多次碰壁，猛然醒悟的时候，我发现 Haskell 就是这样一种邪教 :p

每一种宗教都有一个神秘的，体现自己“精神”的徽标。Haskell 的是这个样子：

当一个初学者进入 Haskell 的 IRC 聊天室的时候，老手们会极其耐心地问答他的问题。他们告诉他，Haskell 社区是最友好，最不宗教，最讲科学和理性的社区。可是久而久之，你发现其实不是那个样子。他们对你友好，当且仅当你没有指出 Haskell 的致命缺陷。如果你知道了那些缺陷，要跟他们讨论的时候，就会发现一切都变了。他们会说，你懂不起！我们是科学家！如果你不承认这一点，我们就杀了你！:p

Haskell 的社区喜欢在他们的概念里省掉“纯”这个字，把 Haskell 叫做“函数式语言”。他们喜欢“纠正”别人的概念。他们告诉人们，“不纯”的函数式语言，其实都不配叫做“函数式语言”。在他们的这种定义下，Lisp 这么老牌的函数式语言，居然都不能叫“函数式语言”了。但是看完这篇文章你就会发现，其实他们的这种定义是狭隘和错误的。

在 Haskell 里面，你不能使用通常语言里面都有的赋值语句，比如 Pascal 里的 `x:=1`，C 和 Java 里的 `x=1`，或者 Scheme 里的 `(set! x 1)`，Common Lisp 里的 `(setq x 1)`。这样一来，你就不可能保留“状态”（state）。所谓“状态”，就是指“随机数种子”那样的东西，其实本质上就是“全局变量”。比如，在 C 语言里定义 `random()` 函数，你可以这么做：

```
int random()
{
  static int seed = 0;
  seed = next_random(seed);
  return seed;
}
```

这里的 `seed` 是一个“static 变量”，其本质就是一个全局变量，只不过这个全局变量只能被 `random` 这一个函数访问。每次调用 `random()`，它都会使用 `next_random(seed)` 生成下一个随机数，并且把 `seed` 的值更新为这个新的随机数。在 `random()` 的执行结束之后，`seed` 会一直保存这个值。下一次调用 `random()`，它就会根据 `seed` 保存的值，算出下一个随机数，然后再次更新 `seed`，如此继续。这就是为什么每一次调用 `random()`，你都会得到不同的随机数。

可是在 Haskell 里面情况就很不一样了。由于 Haskell 不能保留状态，所以同一个“变量”在它作用域的任何位置都具有相同的值。每一个函数只要输入相同，就会输出同样的结果。所以在 Haskell 里面，你不能轻松的表达 `random` 这样的“不纯函数”。为了让 `random` 在每次调用得到不同的输出，你必须给它“不同的输入”。那怎么才能给它不同的输入呢？Haskell 采用的办法，就是把“种子”作为输入，然后返回两个值：新的随机数和新的种子，然后想办法把这个新的种子传递给下一次的 `random` 调用。所以 Haskell 的 `random` 的“线路”看起来像这个样子：

```
（旧种子）---> （新随机数，新种子）
```

现在问题来了。得到的这个新种子，必须被准确无误的传递到下一个使用 `random` 的地方，否则你就没法生成下一个随机数。因为没有地方可以让你“暂存”这个种子，所以为了把种子传递到下一个使用它的地方，你经常需要让种子“穿过”一系列的函数，才能到达目的地。种子经过的“路径”上的所有函数，必须增加一个参数（旧种子），并且增加一个返回值（新种子）。这就像是用一根吸管扎穿这个函数，两头通风，这样种子就可以不受干扰的通过。

所以你看到了，为了达到“纯函数”的目标，我们需要做很多“管道工”的工作，这增加了程序的复杂性和工作量。如果我们可以把种子存放在一个全局变量里，到需要的时候才去取，那就根本不需要把它传来传去的。除 `random()` 之外的代码，都不需要知道种子的存在。

为了减轻视觉负担和维护这些进进出出的“状态”，Haskell 引入了一种叫 monad 的概念。它的本质是使用类型系统的“重载”（overloading），把这些多出来的参数和返回值，掩盖在类型里面。这就像把乱七八糟的电线塞进了接线盒似的，虽然表面上看起来清爽了一些，底下的复杂性却是不可能消除的。有时候我很纳闷，在其它语言里易如反掌的事情，为什么到 Haskell 里面就变成了“研究性问题”，很多时候就是 monad 这东西在捣鬼。特别是当你有多个“状态”的时候，你就需要使用像 monad transformer 这样的东西。而 monad transformer 在本质上其实是一个丑陋的 hack，它并不能从根本上解决问题，却可以让你伤透脑筋也写不出来。有些人以为会用 monad 和 monad transformer 就说明他水平高，其实这根本就是自己跟自己过不去而已。

当谈到 monad 的时候，我喜欢打这样一个比方：

> 使用含有 monad 的“纯函数式语言”，就像生活在一个没有电磁波的世界。
>
> 在这个世界里面没有收音机，没有手机，没有卫星电视，没有无线网，甚至没有光！这个世界里的所有东西都是“有线”的。你需要绞尽脑汁，把这些电线准确无误的通过特殊的“接线器”（monad）连接起来，才能让你的各种信息处理设备能够正常工作，才能让你自己能够看见东西。如果你想生活在这样的世界里的话，那就请继续使用 Haskell。

其实要达到纯函数式语言的这种“纯”的效果，你根本不需要使用像 Haskell 这样完全排斥“赋值语句”的语言。你甚至不需要使用 Lisp 这样的“非纯”函数式语言。你完全可以用 C 语言，甚至汇编语言，达到同样的效果。

我只举一个非常简单的例子，在 C 语言里面定义如下的函数。虽然函数体里面含有赋值语句，它却是一个真正意义上的“纯函数”：

```
int f(int x) {
    int y = 0;
    int z = 0;
    y = 2 * x;
    z = y + 1;
    return z / 3;
}
```

这是为什么呢？因为它计算的是数学函数 `f(x) = (2x+1)/3` 。你给它同样的输入，肯定会得到同样的输出。函数里虽然对 `y` 和 `z` 进行了赋值，但这种赋值都是“局部”的，它们不会留下“状态”。所以这个函数虽然使用了被“纯函数程序员”们唾弃的赋值语句，却仍然完全的符合“纯函数”的定义。

如果你研究过编译器，就会理解其中的道理。因为这个函数里的 `y` 和 `z`，不过是函数的“数据流”里的一些“中间节点”，它们的用途是用来暂存一些“中间结果”。这些局部的赋值操作，跟函数调用时的“参数传递”没有本质的区别，它们不过都是把信息传送到指定的节点而已。如果你不相信的话，我现在就可以把这些赋值语句全都改写成函数调用：

```
int f(int x) {
    return g(2 * x);
}

int g(int y) {
    return h(y + 1);
}

int h(int z) {
    return z/3;
}
```

很显然，这两个 f 的定义是完全等价的，然而第二个定义却没有任何赋值语句。第一个函数里对 `y` 和 `z` 的“赋值语句”，被转换成了等价的“参数传递”。这两个程序如果经过我写的编译器，会生成一模一样的机器代码。所以如果你说赋值语句是错误的话，那么函数调用也应该是错误的了。那我们还要不要写程序了？

盲目的排斥赋值语句，来自于对“纯函数”这个概念的片面理解。很多研究像 Haskell，ML 一类语言的专家，其实并不明白我上面讲的道理。他们仿佛觉得如果使用了赋值，函数就肯定不“纯”了似的。CMU 的教授 Robert Harper 就是这样一个极端。他在一篇博文里指出，人们不应该把程序里的“变量”叫做“变量”，因为它跟数学和逻辑学里所谓的“变量”不是一回事，它可以被赋值。然而，其果真如他所说的那样吗？如果你理解了我对上面的例子的分析，你就会发现其实程序里的“变量”，跟数学和逻辑学里面的“变量”相比，其实并没有本质的不同。

程序里的变量甚至更加严格一些。如果你把数学看作一种程序语言的话，恐怕没有一本数学书可以编译通过。因为它们里面充满了变量名冲突，未定义变量，类型错误等程序设计的低级错误。你只需要注意概率论里表示随机数的大写变量（比如 X），就会发现数学所谓的“变量”其实是多么的不严谨。这变量 X 根本不需要被赋值，它自己身上就带“副作用”！实际上，90%以上的数学家都写不出像样的程序来。所以拿数学的“变量”来衡量程序语言的“变量”，其实是颠倒了。我们应该用程序的“变量”来衡量数学的“变量”，这样数学的语言才会有所改善。

逻辑学家虽然有他们的价值，但他们并不是先知，并不总是对的。由于沉迷于对符号的热爱，他们经常看不到事物的本质。虽然他们理解很多符号公式和推理规则，但他们却经常不明白这些符号和推理规则，到底代表着自然界中的什么物体，所以有时候他们连最基本的问题都会搞错（比如他们有时候会混淆“全称量词”∀的作用域）。逻辑学家们的教条主义和崇古作风，也许就是图灵当年在 Church 手下做学生那么孤立，那么痛苦的原因。也就是这个图灵，在某种程度上超越了 Church，把一部分人从逻辑学的死板思维模式下解放了出来，变成了“计算机科学家”。当然其中某些计算机科学家堕入了另外一种极端，他们对逻辑学已有的精华一无所知，所以搞出一些完全没有原则的设计，然而这不是这篇文章的主题。

所以综上所述，我们完全没有必要追求什么“纯函数式语言”，因为我们可以在不引起混淆的前提下使用赋值语句，而写出真正的“纯函数”来。可以自由的对变量进行赋值的语言，其实超越了通常的数理逻辑的表达能力。如果你不相信这一点，就请想一想，数理逻辑的公式有没有能力推断出明天的天气？为什么天气预报都是用程序算出来的，而不是用逻辑公式推出来的？所以我认为，程序其实在某种程度上已经成为比数理逻辑更加强大的逻辑。完全用数理逻辑的思维方式来对程序语言做出评价，其实是很片面的。

说了这么多，对于“函数式语言”这一概念的误解，应该消除得差不多了。其实“函数式语言”唯一的要求，应该是能够在任意位置定义函数，并且能够把函数作为值传递，不管这函数是“纯”的还是“不纯”的。所以像 Lisp 和 ML 这样的语言，其实完全符合“函数式语言”这一称号。