关于语言的思考

之前写了那么多 Haskell 的不好的地方，却没有提到它好的地方，其实我必须承认我从 Haskell 身上学到了非常重要的东西，那就是对于“类型”的思考。虽然 Haskell 的类型系统有过于强烈的约束性，从一种“哲学”的角度（不是数学的角度）来看非常“不自然”，但如果一个程序员从来没学过 Haskell，那么他的脑子里就会缺少一种重要的东西。这种东西很难从除 Haskell，ML，Clean，Coq，Agda 以外的其它语言身上学到。

Haskell 给我的启发

一个没有学过 Haskell 的 Scheme 程序员最容易犯的一个错误就是，把除 `#f`（Scheme 的逻辑“假”） 以外的任何值都作为 `#t`（Scheme 的逻辑“真”）。很多人认为这是 Scheme 的一个“特性”，可是殊不知这其实是 Scheme 的极少数缺点之一。如果你了解 Lisp 的历史，就会发现在最早的时候，Lisp 把 `nil`（空链表）这个值作为“假”来使用，而把 `nil` 以外的其它值都当成“真”。这带来了逻辑思维的混乱。

Scheme 对 Lisp 的这种混乱做法采取了一定的改进，所以在 Scheme 里面，空链表 `'()` 和逻辑“假”值 `#f` 被划分开来。这是很显然的事情，一个是链表，一个是 `bool`，怎么能混为一谈。Lisp 的这个错误影响到了很多其它的语言，比如 C 语言。C 语言把 0 作为“假”，而把不是 0 的值全都作为“真”。所以你就看到有些自作聪明的 C 程序员写出这样的代码：

```
int i = 0;
...
...
if (i++) { ...}
```

Scheme 停止把 `nil` 作为“假”，却仍然把不是 `#f` 的值全都作为“真”。Scheme 的崇拜者一般都告诉你，这样做的好处是，你可以使用

```
(or x y z)
```

这样的表达式，如果其中有一个不是 `#f`，那么这个表达式会直接返回它实际的值，而不只是 `#t`。然后你就可以写这样的代码：

```
(cond
 [(or x y z)
  => (lambda (found)
       (do-something-with found))])
```

而不是：

```
(let ([found (first-non-false x y z)])
  (cond
   [(not (eq? found #f))
    (do-something-with found)]))
```

第一段代码使用了 Scheme 的一个特殊“语法”，`=>` 后面的 `(lambda (found) ...)` 会把 `(or x y z)` 返回的值作为它的参数 `found`，然后返回函数计算出的结果。第二段代码没有假设任何不是 `#f` 的值都是“真”，所以它不把 `(or x y z)` 放进 `cond` 的条件里，而是首先把它返回的值绑定到 `found`，然后再把这个值放进 `cond` 的条件。

这第二段代码比第一段代码多了一个 let，增加了一层缩进，貌似更加复杂了，所以很多人觉得把不是 `#f` 的值全都作为“真”这一做法是合理的。其实 Scheme 为了达到这个目的，恰好犯了“片面追求短小”的语言设计的小聪明（参考这篇博文）。为了让这种情况变得短小而损失类型的准确，这种代价是非常不值得的。

Haskell 的类型系统就是帮助你严密的思考类似关于类型的问题的。如果你从来没学过 Haskell，你就不会发现这里面其实有个类型错误。可是 Haskell 做得过分了一点，由于对类型推导，一阶逻辑和 category theory 等理论的盲目崇拜，Haskell 里面引入了很多不必要的复杂性。

各种各样的类型推导我设计过不下十个，其中有一些比 Haskell 强大很多。category theory 其实也不是什么特别有用的东西。很多数学家把它叫做“abstract nonsense”，就是说它太“通用”了，以至于相当于什么都没说。我曾经在一个晚上看完了整本的 category theory 教材，发现里面的内容我其实通过自己的动手操作（实现编译器，设计类型系统和静态分析等等），早就明白了。这里面的理论并不能带来对程序语言的简化。恰恰相反，它让程序语言变得复杂。

我对 Haskell 程序员的“天才态度”也感到厌倦，所以我不想再使用 Haskell，然而我的脑子里却留下了它“启发”我的东西。对 Haskell 的理解，让我成为了一个更好的 Scheme 程序员，更好的 Java 程序员，更好的 C++ 程序员，甚至更好的 shell 脚本程序员。我能够在任何语言里再现 Haskell 的编程方式的精髓。然而让我继续用 Haskell ，却就像是让我坐牢一样。本来很简单的事情，到 Haskell 里面就变成一些莫名其妙的新术语。Haskell 的设计者们的论文我大部分都看过，几分钟之内我就知道他们那一套东西怎么变出来的，其实里面很少有新的东西。大部分是因为

Haskell 引入的那些“新概念”（比如 monad）而产生的无须有的问题。世界上有比他们更聪明的人，更简单却更强大的理论。所以不要以为 Haskell 就是世界之巅。

怎么说呢，我觉得每个程序员的生命中都至少应该有几个月在静心学习 Haskell。学会 Haskell 就像吃几天素食一样。每天吃素食显然会缺乏全面的营养，但是每天都吃荤的话，你恐怕就永远意识不到身体里的毒素有多严重。

专攻一门语言的害处

我曾经对人说 C++ 里面其实有一些好东西，但是我没有说的是，C++ 里面的坏东西实在太多了。C++是一门“毒素”很多的语言，就像猪肉一样。

有些人从小写 C++，一辈子都在写 C++，就像每天每顿吃猪肉一样。结果是他们对 C++ 里面的“珍珠”掌握的非常牢靠，以至于出现了一种“脑残”的现象——他们没法再写出逻辑清晰的程序。（这里“珍珠”是一个特殊的术语，它并不含有赞美的意思。请参考这篇博文。）

比如，很多 C++ 程序员很精通 functor 的写法，可是其实 functor 只是由于 C++ 没有 first-class function 而造成的“变通”。C++ 的 functor 永远也不可能像 Scheme 的 lambda 函数一样好用。因为每次需要一个 functor 你都得定义一个新的 class，然后制造这个 class 的对象。如果函数里面有自由变量，那么这些自由变量必须通过构造函数放进 functor 的 field 里面，这样当 functor 内部的“主方法”被调用的时候，它才能知道自由变量的值。所以为此，你又得定义一些 field。麻烦了这么久，你得到的其实不过是 Scheme 程序员用起来就像呼吸空气一样的 lambda。

很多精通 functor 的 C++ 程序员认为会用 functor 就说明自己水平高。殊不知 functor 这东西不但是一个“变通”，而且是从函数式语言里面“学”过来的。在最早的时候，C++ 程序员其实是不知道 functor 这东西的。如果你考一下古就会发现，C++ 诞生于 1983 年，而 Scheme 诞生于 1975 年，Lisp 诞生于 1958 年。C++ 的诞生比 Scheme 整整晚了8年，然而 Scheme 一开始就有 lexical scoping 的 lambda。functor 只不过是对 lambda 的一种绕着弯的模仿。实际上 C++ 后来加进去的一些东西（包括 boost 库），基本上都是东施效颦。

记得2011年11月11日的良辰吉日，C++ 的创造者 Bjarne Stroustrup 在 Indiana 大学做了一个演讲，主题是关于 C++11 的新特性。当时我也在场，主持人 Andrew 是 boost 库的首席设计师之一（他后来有段时间当过我的导师）。他连夸 Stroustrup 会选日子，只遗憾演讲时间没有定在11点。

虽然我对 Stroustrup 的幽默感和谦虚的态度感到敬佩，但我也看出来 C++11 相对于像 Scheme 这样的语言，其实没有什么真正的“新东西”。大部分时候它是在改掉自己的一些坏毛病，然后向其它语言学习一些东西，然后把这些学习的痕迹掩盖起来。可是到最后，它仍然不可能达到其他语言那么原汁原味的效果。然而，由于 C++ 的普及程度高，现成的代码又多，它的地位和重要性还是一时难以动摇的。所以这些“先辈的罪”，我们恐怕要用好几代人的工作才能弥补。

那么 C++ 有什么其他语言没有的好东西呢？其实非常少。我还是有空再讲吧。

多学几种语言

我今天想说其实就是，没有任何一种语言值得你用毕生的精力去“精通”它。“精通”其实代表着“脑残”——你成为了一个高效的机器，而不是一个有自己头脑的人。你必须对每种语言都带有一定的怀疑态度，而不是完全的拥抱它。每个人都应该学习多种语言，这样才不至于让自己的思想受到单一语言的约束，而没法接受新的，更加先进的思想。这就像每个人都应该学会至少一门外语一样，否则你就深陷于自己民族的思维方式。有时候这种民族传统的思想会让你深陷无须有的痛苦却无法自拔。