Hindley-Milner 类型系统的根本性错误

之前的一个时间，我曾经公开过这样一段幻灯片，它是2012年10月的时候，我在 Indiana 大学做的最后一次演讲。由于当时的委婉，我并没有直接说出这些结论的重要性。其实它揭示了 ML 和 Haskell 这类基于 Hindley-Milner 类型系统的语言的一个根本性的错误。

这个错误来源于对一阶逻辑的“全称量词”（universal quantifier，通常写作∀）与程序函数之间关系的误解。在 HM 系统里面，多态（polymorphic）的函数能够被推导为含有全称量词的类型，比如 `\x->x` 的类型被推导为 `∀a.a->a`，但 HM 系统决定这个全称量词的位置的方式，却是没有原则的。这就导致了类型变量（type variable）的作用域（scope）的偏差。

我的研究显示，这个错误来源于 HM 系统最初的一项重要的设计，叫做 let-polymorphism。如果右边的函数是一个多态的函数，比如：

```
let f = \x->x in
  ...
```

let-polymorphism 总是会把全称量词的位置确定在 `let` 的“=”右边。然而这是一个非常错误的做法，它的错误程度近似于早期的 Lisp 所采用的 dynamic scoping。这样做的结果是，全称量词的位置会随着程序的“格式”，而不是程序的“结构”，而变化。至于什么是 dynamic scoping，你可以参考我的这篇博文。

为了弥补这个错误，30多年来，许多的人发表了许多的论文，提出了很多的“改进措施”，比如 value restriction，MLF，等等。但是我的研究却显示，所有这些“改进措施”都是丑陋的 hack。因为他们没有看到问题的根源，所以他们的方案只对一些特殊情况起作用，而不能通用。为此，我可以轻而易举的写出正确的程序，而让它不能通过这些类型系统的检查，比如像我这篇英文博文所示。如果你看到了问题的根源，就会发现 HM 系统的这个错误是无法弥补的，因为它触及了 HM 系统的根基。为了根治这个问题，let-polymorphism 必须被去除掉。

我为此提出了自己的解决方案：在 lambda 的位置进行“generalization”，也就是说把 ∀ 放在 `lambda 的位置`，而不是 `let`。这样一来 let-polymorphism 就不存在了。但是这样一来，HM 系统就不再是 HM 系统，因为它的“模块化类型推导”的性质，就会名存实亡。由于类型里面含有程序的“控制结构”，这个类型系统表面上看起来是在进行“模块化类型检查”，而本质上是在做一个“跨过程静态检查”（interprocedual static analysis）。也就是说，模块化的类型推导，在 HM 这样的没有“类型标记”的体系下，其实是不可能实现的。

为了达到完全通用的模块化类型检查，却又允许多态函数的存在，我们终究会需要在函数的参数位置手工写上类型，这样我们就完全的丧失了 HM 系统设计的初衷。