惰性求值

从之前的几篇博文里面你也许已经看到了，Haskell 其实是问题相当严重的语言，然而这些问题却没有引起足够的重视。我能看到的 Haskell 的问题在于：

- 复杂的基于缩进的语法，使得任何编辑器都不能高效的编辑 Haskell 程序，并且使得语法分析难度加倍。对这个观点，请参考我的博文《谈语法》以及我的英文博文《Layout Syntax Considered Harmful》。
- “纯函数式”的语义以及 monad 其实不是好东西。对此请参考博文《对函数式语言的误解》。
- Haskell 所用的 Hindley-Milner 类型系统，其实含有一个根本性的错误。对此请参考《Hindley-Milner 类型系统的根本性错误》。
- Haskell 所用的 type class，其实跟一般语言（比如 Java）里面的重载（overloading）并没有本质区别。你看到的区别都是因为 Hindley-Milner 系统和重载混合在一起产生的效果。type class 并不能比其它语言里的重载做更多的事。

这样一来，好像 Haskell 的“特征”，要么是错误的，要么就不是自己的。可是现在我再给它加上一棵稻草：Haskell 的惰性求值（lazy evaluation）方式，其实大大的限制了它的运行效率，并且使得它跟并行计算的目标相矛盾。

这是一个对我已经非常明显的问题，所以我只简要的说明一下。惰性求值的方式，使得我们在“需要”一个变量的值的时候，总是有两种可能性：1）这个变量在这之前已经被求值，所以可以直接取值 2）这个变量还没有被求值，也就是说它还是一个 thunk，我们必须启动对它的求值。

可能你已经发现了，这其实带来了类型系统的混乱。任何类型，不管是 Int, Bool, List, … 或者自定义数据类型，都多出了这么一个东西：thunk。它表示的是“还没有求值的计算”。Haskell 程序员一般把它叫做“bottom”，写作 _|_。它的意思是：死循环。因为任何 thunk 都有可能 1）返回一个预定的类型的值，或者 2）导致死循环。

这有点像 C++ 和 Java 里的 null 指针，因为 null 可以被作为任何其他类型使用，却又不具有那种类型的特征，所以会产生意想不到的问题。_|_ 给 Haskell 带来的问题没那么严重，但却一样的不可预料，难以分析和调试。对于 Haskell 来说，有可能出现这样的事情：明明写了一个很小的函数，觉得应该不会花很多时间。结果呢，因为它对某个变量取值，间接的触发了一段很耗时间的代码，所以等了老半天还没返回。想知道是哪里出了问题，却难以发现线索，因为这函数并没有直接或者间接的调用那段耗时间的代码，而是这个变量的 thunk 启动了那段代码。这就导致了程序的效率难以分析：被“惰性”搁在那里的计算，有可能在出乎你意料的地方爆发。这就是所谓“平时不烧香，临时抱佛脚。”

这种不确定性，并没有带来总体计算开销的增加。然而“惰性”却在另外一方面带来了巨大的开销，这就是“问问题”的开销。每当看到一个变量，Haskell 都会问它一个问题：“你被求值了没有？”即使这变量已经被求值，而且已经被取值一百万次，Haskell 仍然会问这个问题：“你被求值了没有？”问一个变量这问题可能不要紧，可是 Haskell 会问几乎所有的变量这个问题，反复的问这个问题。这就累积成了巨大的开销。跟我在另一篇博文里谈到的“解释开销”差不多，这种问题是“运行时”的，所以没法被编译器“优化”掉。

具有讽刺意味的是，Haskell 这种“纯函数式语言”的惰性求值所需要的 thunk，全都需要“副作用”才可以更新，所以它们必须被放在内存里面，而不是寄存器里面。如果你理解了我写的《对函数式语言的误解》，你就会发现连 C 程序里面的“副作用”也没有 Haskell 这么多。这样一来，处理器的寄存器其实得不到有效的利用，从而大大增加了内存的访问。我为什么可以很确信的告诉你这个呢？因为我曾经设计了一个寄存器分配算法，于是开会的时候我问 GHC 的实现者们，你们会不会对一个新的寄存器分配算法感兴趣，我可以帮你们加到 GHC 里面。结果他们说，我们不需要，因为 Haskell 到处都是 thunk，根本就没什么机会用寄存器。

所以，问太多问题，没法充分利用寄存器，这使得 Haskell 在效率上大打折扣。

然后我们来看看，为什么惰性求值会跟并行计算的目标相冲突。这其实很明显，它的原因就在于“惰性求值”的定义。惰性求值说：“到需要我的时候再来计算我。”而并行计算说：“到需要你的时候，你最好已经被某个处理器算出来了。”所以你看到了，并行计算要求你“勤奋”，要求你事先做好准备。而惰性求值本来就是很“懒”，怎么可能没事找事，先把自己算出来呢？由于这个问题来自于“惰性求值”的定义，所以这是不可调和的矛盾。

所以，惰性求值不管是在串行处理还是在并行处理的时候，都会带来效率上的大打折扣，它是一个很鸡肋的语言特征。

虽然惰性求值不能给我们带来直接的益处，但它背后的理论思想却可以启发另外的设计。如果你想真的了解惰性求值的原理，可以先看一下我写的一个惰性求值的解释器。看看如何在不到 40 行代码之内，实现 Haskell 语义的精髓：

https://github.com/yinwang0/lightsabers/blob/master/interp-lazy.rkt