Chez Scheme 的传说

在上一篇博文的最后，我提到了 Lisp 编译器的问题。由于早期的 Lisp 编译器生成的代码效率普遍低下，成为了 Lisp 失败的主要原因之一。而现在的高性能 Lisp 编译器（比如 Chez Scheme），其实已经可以生成非常高效的代码，甚至可以匹敌 C 程序的速度。如果你看得到我脑子里的东西，就会明白这完全不是吹牛，而是科学的结论。我在这里介绍一下我写 Scheme 编译器的经历，也许你就会从根本上明白为什么我会对此这么自信。这里的介绍其实不止针对函数式语言，而且针对所有语言的编译器。

编译器是一种神秘，有趣，又无聊的的程序。说它神秘，是因为只有非常少的人知道如何写出优秀的编译器。这些会写编译器的人，就像身怀绝技的武林高手一样神出鬼没。说它有趣，是因为编译器的技术里面含有大量的“哲学问题”和深刻的理论（比如 partial evaluation）。但为什么又说它无聊呢？因为你一旦掌握了编译器技术里面最精华的原理，就会发现其实说来说去就那么点东西。编译器代码里面的“创造性含量”其实非常低。里面有些固定的“模式”，几十年都不变。这是因为编译器只是一种“工具”，而不是最终的“目的”。它就像做菜的锅一样，只有屈指可数的那几种形状。设计应用程序才是程序员的最终目的。只有应用程序才能有无穷无尽的创造性。这就像厨师用同样的锅，却能做出无穷变化的菜肴来。然而，我并不是说普通程序员不应该学习写编译器。相反，编译器的原理是非常重要的知识。不理解编译原理的应用程序设计者，就像不理解菜锅组成原理的厨师。

先来说一说为什么早期的 Lisp 编译器生成的代码效率低下吧。在函数式语言的早期，由于它比普通的语言多了一些表达力强大的构造（比如函数作为值传递），人们其实都不知道如何实现它的编译器。很多 Scheme 的编译器其实只是把 Scheme 编译成 C，然后再调用 C 语言的编译器。Haskell 的编译器 GHC 在早期也是这样的。而且由于 C 编译器生成的汇编代码不完全符合 Haskell 的需求，GHC 里面含有一个 Perl 脚本，专门用于调整这汇编代码的结构。这个 Perl 脚本，由于它的工作方式毫无原则，被叫做 evil mangler。现在这个东西已经被去掉了，但从它曾经的存在你可以看出，其实函数式编译器的技术在早期是相当混沌的。

在我看来，早期 Lisp 编译器出现的主要问题，其实在于对编译的本质的理解，以及编译器与解释器的根本区别。解释器之所以大部分时候比编译器慢，是因为解释器“问太多的问题”。每当看到一个构造，解释器就会问：“这是一个整数吗？”“这是一个字符串吗？”“这是一个函数吗？”…… 然后根据问题的结果进行不同的处理。这些问题，在编译器的理论里面叫做“解释开销”（interpretive overhead）。编译的本质，其实就是在程序运行之前进行“静态分析”，试图一劳永逸的回答这些问题。于是编译后的代码根本不问这种问题，它直接就知道那个位置肯定会出现什么构造，应该做什么事，于是它就直接去做了。早期的 Lisp 编译器，以及现在的很多 Scheme 编译器出现的问题其实在于，它们并没有干净的消除这些问题，甚至根本没有消除这些问题。

当我最早学习 Scheme 语言的时候，我发现 Scheme 有太多的“实现”：PLT Scheme（现在叫 Racket），MIT Scheme, Scheme 48, Bigloo, Chicken, Gambit, Guile, … 让人搞不清楚哪一个更好。有些 Scheme 实现显得高级一些，但实际用起来总是感觉不放心，因为你心里总想着，这代码编译出来到底能不能跟 C 语言代码比？这也是我后来开始使用 Common Lisp 的原因，因为 Common Lisp 似乎有挺多高效的编译器（CMUCL，Lispworks，Allegro 等等）。

直到有一天，我发现了 Chez Scheme，它改变了我对 Scheme 编译器，以至于整个编译器概念的理解。当时我只下载了 Chez Scheme 的免费版本，叫做 Petite。Petite 与正式版 Chez Scheme 的区别是，它不输出二进制代码，所以你不能把编译后的代码拿去销售。另外出于商业目的，Petite 的出错信息非常的“简约”，以至于有时候你不得不用其它的 Scheme 实现，才能找到 bug 的位置。但是一运行就见分晓，Petite 被作为一个“解释器”直接运行 Scheme 代码，比其他的 Scheme 实现编译后的代码还要快很多倍。

Chez Scheme 导致了我命运的改变，我怎么也没有想到，自己最终会见到它的作者 R. Kent Dybvig，并且成为他的学生。我只能说也许一切都是天意吧。第一次见到 Kent 的时候，他安静的对我说，你应该拥有自己的代码，将来有一天，你会发现它的价值。

也就是这个 Kent，单枪匹马的创造了 Chez Scheme，世界上唯一的商业 Scheme 编译器，并且为此成立了自己的公司（Cadence Research Systems）。Chez Scheme 价格不菲，而且不明码实价，它的价格跟项目的大小和公司的规模成正比。有些大公司花重金购买 Chez Scheme 用于一些核心的项目。其中有些公司为了保证这编译器的安全，又花了好几倍的价钱买下了它的源代码。Kent 的公司只有他一个人，不用操心管理，也不用操心销售。所以他过的非常舒服，基本是一个不愁吃穿，不问世事的人。

Kent 是我一生中见过的最神秘，最酷的人。他几乎从来不表扬任何人，但也不贬低任何人。从冷漠的言语之中，你仿佛感觉他并不是这个世界上的人。任何人的喜怒与哀乐，傲慢与偏见，蔑视与奉承，全都不能引起他情绪的变化。他的心里有许许多多的秘密，你需要一些技巧才能套出他的真言。他很少发表论文，却把别人的论文全都看得很透。没有人知道他的核心技术，他也从来不在乎别人是否了解他的水平。最让人惊奇的是，没有人知道他叫什么名字！他的全名叫 R. Kent Dybvig，那么 R. 就应该是他的 first name。然而，却从来没有人知道那个 R. 是哪一个名字的简写，所以大家只好叫他的 middle name，Kent。

Chez Scheme 生成的“目标代码”效率之高，我还没有见到任何其它 Scheme 编译器可以与之匹敌。而它的“编译速度”之快，没有任何语言的任何编译器可以相提并论（注意我去掉了“Scheme”这个限定词）。Chez Scheme 可以在 5 秒钟之内完成从头到尾的自我编译。想想编译 GCC 或者 GHC 需要多少时间，你就明白差距了。

另外值得一提的是，Chez Scheme 从头到尾都是 Kent 一个人的作品。它的工作原理是从 Scheme 源程序一直编

译到机器代码，而不依赖任何其他语言的编译器。它甚至不依赖第三方的汇编器，所有三种体系构架（Intel, ARM, SPARC）的汇编器，都是 Kent 自己写的。为什么这样做呢？因为几乎没有其它人的编译器代码能够达到他的标准。连 Intel 自己给自己的处理器写的汇编器，都不能满足他的要求。

如果你上了 Kent 的课，再来看看普通的编译器书籍（比如有名的 Dragon Book），或者 LLVM 的代码，你就会发现 Kent 的水平其实远在这些知名的大牛之上。我为什么可以这么说呢？因为如果你的水平不如这些人的话，你自己都会对这种判断产生怀疑。而如果你超过了别人，他们的一言一行，他们的每一个错误，都像是处于你的显微镜底下，看得一清二楚。这就是为什么有一天我拿起 Dragon Book，感觉它变得那么的幼稚。而其实并不是它变幼稚了，而是我变成熟了。实话实说吧，在编译器这个领域，我觉得 Kent 很有可能就是世界的 No.1。

如果你不了解 Scheme 的编译器里面有什么东西，也许就会轻视它的难度。Scheme 是比 C 语言高级很多的语言，所以它的编译器需要做比 C 语言的编译器多很多的事情。在 Kent 的编译器课程的前半段，我们其实本质上是在实现一个 C 语言的编译器，把一种基于“S表达式”的中间语言，编译为 X64 汇编代码。在后半学期的课程中，我们才加入了各种 Scheme 的先进功能，比如函数作为值（需要进行 closure conversion 以及 closure 优化），尾递归优化（tail-call optimization），等等。另外，我还自己为它加入了一种非常漂亮的技术，叫做 online partial evaluation。这种技术可以在一个 pass 就完成普通编译器需要好几个 pass 才能完成的优化。

在这些先进的优化技术之下，几乎所有的冗余代码都会被编译器消除掉。这些优化的智能程度，在很多方面拥有人类思维没法达到的准确性和深度。如果你的程序没有使用到 Scheme 特有的功能，那么生成的目标代码就会跟 C 语言编译后的代码没有什么两样。比如，如果你的代码没有把函数作为值传递，或者你的函数里面没有“自由变量”，或者你的函数里虽然有自由变量，但是你却没法在函数外部改变它的值，那么生成的代码里面就不会含有“闭包”，也就不会产生多余的内存数据交换。你有时甚至会得到比 C 程序编译之后更好的代码，因为我们的“后端”编译器其实比 GCC，LLVM 之类的 C 编译器先进。

Kent 的课程编译器有很好的结构，它被叫做“nanopass 编译器构架”。它的每一个 pass 只做很小的一件事情，然后这些 pass 被串联起来，形成一个完整的编译器。编译的过程，就是将输入程序经过一系列的变换之后，转化为机器代码。你也许发现了，这在本质上跟 LLVM 的构架是一样的。但是我可以告诉你，我们的课程编译器比 LLVM 干净利落许多，处于远远领先的地位。每一节课，我们都学会一个 pass。每一个讲义，都非常精确的告诉你需要干什么。每一次的作业，提交的时候都会经过上百个测试（当然 Kent 不可能把 Chez Scheme 的测试都给我们），如果没有通过就会被拒绝接受。这些测试也可以下载，用于自己的调试。有趣的是，每一次作业我们都需要提交一些自己写的新测试，目的是用于“破坏”别人的编译器。所以我们每次都会想出很刁钻的输入代码，让同学的日子不好过。当然是开玩笑的，这种做法其实大大的提高了我们对编译器测试的理解和兴趣，以及同学之间的友谊。这比起我曾经在 Cornell 选过（然后 drop 掉）的编译器课程，真是天壤之别。

在课程的最后，我们做出了一个完整的编译器，它可以把 Scheme 最关键的子集编译到 X64 汇编代码，然后通过 GNU 的汇编器转化成机器代码。在最后的一节课，Kent 对我们的学期做了一个令人难忘的总结。他说：“你们现在写出的这个编译器里面含有很多先进的技术。也许过一段时间再回头看这段代码，你们才会发现它的价值。如果你们觉得自己已经成为了编译器的专家，那我就告诉你们，你们提交的最快的编译器，编译速度比 Chez Scheme 慢了 700 倍。但是不要灰心，我告诉你们哪些地方可以改进……”

只有极少数的人见到过 Chez Scheme 的源代码，我也没有看见过。但是见到过它的人告诉我，Chez Scheme 里面其实只有很少几个 pass，而不是像我们的课程编译器有 50 个左右的 pass，这节省了很多用于“遍历”代码树所需要的时间。Chez Scheme 只使用了一些非常简单的算法，没有使用论文里很炫很复杂的方法，这也是它速度快的原因之一。比如它的寄存器分配，没有使用通常的“图着色”（graph coloring）方法，而是使用非常简单的一种类似 linear scan 的算法，生成的代码效率却更高。另外，Scheme 使用“S表达式”作为它的语法，使得“语法分析”的速度非常之快。其它语言由于使用了复杂的语法，挺大一部分编译时间其实花在了语法分析上面。

所以实际上 Chez Scheme 早就有了超越世人的技术，Kent 却很少为它们发表论文。这是因为他自私吗？应该不是。他已经通过他的课程给予了我们那么宝贵的礼物，我们又怎能要求更多？所以对于更深入的内容，我都是自己摸索出解决方案，再去套他的口气，看他有没有更好的想法。于是有时候我会很惊讶的发现他的一些非常透彻的见解。比如有一天我问他，为什么编译器需要进行寄存器分配？为什么需要寄存器？我觉得 Knuth 设计的 MMIX 处理器里的“寄存器环”，也许能够从根本上避免“寄存器分配”这问题。他听了之后不动声色的说，MMIX 的寄存器环（以及 SPARC 的寄存器窗口）其实是有问题的，当函数递归调用达到一定的深度之后，寄存器环里有再多寄存器都会被用光，到时候就会出现大量的寄存器与内存之间的数据交换，而被“压栈”之后的寄存器，并不会得到有效地“再利用”。于是我才发现，他不但早已了解 MMIX 的设计，而且看透了它的本质。

有趣的是在课程进行之中的时候，我发现自己有些突发灵感的做法，其实已经超越了 Chez Scheme，以至于在某些 pass 会生成比它还要高效的代码，然而我的编译器代码却比它的还要短小（当然绝大部分时间我的代码不如 Chez Scheme）。于是我就隐约的发现，Kent 有时候会悄悄的花时间看我的作业，想搞明白我是怎么做的，但却不想让我知道。有一天开会的时候 Kent 没有来，他的编译器课程助教 Andy 对我说：“Kent 还在对你写的代码进行一些侦探工作……” 从任何人那里得到启发，吸收并且融入到自己的能力里面，也许就是 Kent 练就如此盖世神功的秘诀吧。

我想，这篇文章就该到此结束了。写这些东西的目的，其实只是树立人们对于函数式语言编译器的信心。它们有些其实比 C 和 C++ 之类语言的编译器高明很多。我没有时间也没有精力去讲述这编译器里面的细节，因为它实在是非常困难，却又非常优雅的程序。如果你有兴趣的话，可以看看我最后的代码。由于版权原因，有些辅助部件我不能放在网上，所以你并不能运行它，只能看一个大概的形状。如果你需要一个 Scheme 版本用于学习的话，Chez Scheme 有一个免费的版本叫做 Petite Chez Scheme，可以免费下载。因为 Petite 的出错信息非常不友好，所以我也推荐 Racket 作为替补。不过你需要注意的是，Racket 的速度比起 Chez Scheme 是天壤之别。