

GTF - Great Teacher Friedman

写小人书的老顽童

Dan Friedman 是 Indiana 大学的教授，程序语言领域的创始人之一。他主要的著作《The Little Schemer》（前身叫《The Little Lisper》）是程序语言界最具影响力的书籍之一。现在很多程序语言界的元老级人物，当年都是看这本“小人书”学会了 Lisp/Scheme，才决心进入这一领域。



Friedman 对程序语言的理解可以说是世界的最高标准，很可惜的是，由于他个人的低调，他受到很多人的误解。很多人以为他只懂得 Scheme 这种“类型系统落后的语言”。有些人觉得他只顾自己玩，不求“上进”，觉得他的研究闭门造车，不“前沿”。我也误解过他，甚至在见面之前，根据这些书的封面，我断定他肯定是个年轻小伙。结果呢，第一次见到他的时候，他已经过了60岁大寿。

程序语言的研究者们往往追逐一些“新概念”，却未能想到很多这些新概念早在几十年前就被 Friedman 想到了。举个例子，Haskell 所用的 lazy evaluation 模型，最早就是他在 1976 年在与 David Wise 合写的论文“CONS should not Evaluate its Arguments”中提出来的。

虽然写了 The Little Schemer，但 Friedman 的学识并不限于 Scheme。他不断地实验各种其它的语言设计，包括像 ML 一类的含有静态类型系统的函数式语言，逻辑式语言，面向对象语言，用于定理证明的语言等等。在每次的试验之后，他几乎都会写一本书，揭示这些语言最精要的部分。

觉得 ML 比 Scheme 先进很多的人们应该看看 Friedman 这本书：The Little MLer：



想要真正理解 Java 设计模式的人可以看看这本：A Little Java, A Few Patterns:



这些东西的优点和弱点仿佛在他心里都有数，他几乎总是指向正确的前进方向。

你知道些什么？

可惜的是，由于个人原因，Friedman 始终没有成为我正式的导师（他“超然物外”，几乎完全不关心自己的学生什么时候能毕业），但他确实是这一生中教会我最多东西的人。所以我想写一些关于他的小故事。也许你能从中看出一个世界级的教育者是什么样子的。我来 IU 之前一位师兄告诉我，Dan Friedman 就像指环王里的甘道夫 (Gandalf)，来了之后发现确实很像。

第一次在办公室见到 Friedman 的时候，他对我说：“来，给我讲讲你知道些什么？”我自豪地说：“我在 Cornell 上过研究生的程序语言课，会用 ML 和 Haskell，看过 Paul Graham 的 On Lisp，Peter Norvig 的 Paradigms of Artificial Intelligence Programming，Richard Gabriel 的一些文章……”他看着我平静地笑了：“不错，你已经有一定基础……”

这么几年以后，我才发现他善良的微笑里面其实隐藏着难以启齿的秘密：当时的我是多么的幼稚。在他的这种循循善诱之下，我才逐渐的明白了，知识的深度是无止境的。他的水平其实远在以上这些人之上，可是出于谦虚，他不能自己把这话说出来。

反向运行

Dan Friedman 已经远远超过了退休年龄，却仍然坚持教学。他的本科生程序语言课程 C311 是 IU 的“星级课程”。我最敬佩的，其实是他那孩子般的好奇心和探索精神。几乎每一年的 C311，他都会发明不同的东西来充实课程内容。有时候是一种新的逻辑编程语言（叫 miniKanren），有时候是些小技巧（比如把 Scheme 编译成 C 却不会堆栈溢出），等等。

Friedman 研究一个东西的时候总是全身心的投入，执着的热爱。自从开始设计一个叫 miniKanren 的逻辑编程语言，Friedman 多了一句口头禅：“Does it run backwards?”（能反向运行吗？）因为逻辑式的语言（像 Prolog）的程序都是能“反向运行”的。普通程序语言写的程序，如果你给它一个输入，它会给你一个输出。但是逻辑式语言很特别，如果你给它一个输出，它可以反过来运行，给你所有可能的输入。但是 Friedman 真的走火入魔了。不管别人在讲什么，经常最后都会被问一句：“Does it run backwards?”让你哭笑不得。

Friedman 有一个本领域的人都知道的“弱点”——他不喜欢静态类型系统 (static type system)。其实 Scheme 专家们大部分都不喜欢静态类型系统。为此，他深受“类型专家”们的误解甚至鄙视，可是他都从容对待之。

有一次在他的进阶课程 B621 上，他给我们出了一道题：用 Scheme 实现 ML 和 Haskell 所用的 Hindley-Milner 类型系统。这种类型系统的工作原理一般是，输入一个程序，它经过对程序进行类型推导 (type inference)，输出一个类型。如果程序里有类型错误，它就会报错。由于之前在 Cornell 用 ML 实现过这东西，再加上来到 IU 之后对抽象解释 (abstract interpretation) 的进一步理解，我很快做出了这个东西，而且比在 Cornell 的时候做的还要优雅。

他知道我做出来了，很高兴的样子，让我给全班同学（也就8, 9个人）讲我的做法。当我自豪的讲完，他问：“Does it run backwards? 如果我给它一个类型，它能自动生成出符合这个类型的程序来吗？”我愣了，欲哭无泪啊，“不能……”他往沙发靠背上一躺，得意的笑了：“我的系统可以！哈哈！我当年写的那个类型系统比你这个还要短呢。我早就知道这些类型系统怎么做，可我就是不喜欢。哈哈哈哈哈……”

我后来对类型系统的进一步研究显示，Hindley-Milner 类型系统确实有很多不必要的问题，才导致了他不喜欢。他就是这样一个人。他喜欢先把你捧上天，再把你打下来，让你知道天外有天 :-)

miniCoq

你永远想象不到 Dan Friedman 的思维的极限在哪里。当你认为他是一个函数式语言专家的时候，他设计了 miniKanren，一种逻辑式编程语言 (logic programming language)，并且写出《The Reasoned Schemer》这样的书，用于教授逻辑编程。当你认为他不懂类型系统的时候，他开始捣鼓当时最热门的 Martin-Löf 类型理论，并且开始设计机器证明系统。而他做这些，完全是出于自己的兴趣。他从来不在乎别人在这个方向已经做到了什么程度，却经常能出乎意料的简化别人的设计。

有一次系里举办教授们的“闪电式演讲”(lightening talk), 每位教授只有5分钟时间上去介绍自己的研究。轮到Friedman的时候, 他慢条斯理的走上去, 说: “我不着急。我只有几句话要说。我不知道我能不能拖够5分钟.....”大家都笑了。他接着说: “我现在最喜欢的东西是 Curry-Howard correspondence 和定理证明。我觉得现在的机器证明系统太复杂了, 比如 Coq 有 nnnnn 行代码。我想在 x 年之内, 简化 Coq, 得到一个 miniCoq.....”

miniCoq... 听到这个词全场都笑翻了。为什么呢? 自己去联想吧。从此, “Dan Friedman 的 miniCoq” 成为了 IU 的程序语言学生茶余饭后的笑话。

但是 Friedman 没有吹牛。他总是说到做到。他已经写出一个简单的定理证明工具叫 JBob (迫于社会舆论压力, 不能叫 miniCoq), 而且正在写一本书叫《The Little Prover》, 用来教授最重要的定理证明思想。他开始在 C311 上给本科生教授这些内容。我看了那本书的初稿, 获益至深, 那是很多 Coq 的教材都不涉及的最精华的道理。它不仅教会我如何使用定理证明系统, 而且教会了我如何设计一个定理证明系统。我对他说: “你总是有新的东西教给我们。每隔两年, 我们就得重新上一次你的课!”

C311

当我刚从 Cornell 转学到 IU 的时候, Dan Friedman 叫我去上他的研究生程序语言课 B521。我当时以自己在 Cornell 上过程序语言课程为由, 想不去上他的课。Friedman 把我叫到他的办公室, 和蔼的对我说: “王垠, 我知道你在 Cornell 上过这种课。我也知道 Cornell 是比 IU 好很多的学校。可是每个老师的教学方法都是不一样的, 你应该来上我的课。我和我的朋友们在这里做教授, 不是因为喜欢这个学校, 而是因为我们的家人和朋友都在这里。”后来由于跟 Amr Sabry (我现在的导师) 的课程 B522 时间重合, 他特别安排我坐在本科生的 C311 的课堂上, 却拿研究生课程的学分。后来发现, 这两门课的内容基本没有区别, 只不过研究生的作业要多一些。

在第一堂课上, 他说了一句让我记忆至今的话: “《The Little Schemer》和《Essentials of Programming Languages》是这门课的参考教材, 但是我上课从来不讲我的书里的内容。”刚开始, 我就发现这门课跟我在 Cornell 学到的东西很不一样。虽然有些概念, 比如 closure, CPS, 我在 Cornell 都学过, 在他的课堂上, 我却看到这些概念完全不同的一面, 以至于我觉得其实我之前完全不懂这些概念! 这是因为在 Cornell 学到这些东西的时候只是用来应付作业, 而在 Friedman 的课上, 我利用它们来完成有实际意义的目标, 所以才真正的体会到这些概念的内涵和价值。

一个例子就是课程进入到没几个星期的时候, 我们开始写解释器来执行简单的 Scheme 程序。然后我们把这个解释器进行 CPS 变换, 引入全局变量作为“寄存器”(register), 把 CPS 产生的 continuation 转换成数据结构(也就是堆栈)。最后我们得到的是一个抽象机 (abstract machine), 而这在本质上相当于一个真实机器里的中央处理器 (CPU) 或者虚拟机 (比如 JVM)。所以我们其实从无到有, “发明”了 CPU! 从这里, 我才真正的理解到寄存器, 堆栈等的本质, 以及我们为什么需要它们。我才真正的明白了, 冯诺依曼体系构架为什么要设计成这个样子。后来他让我们去看一篇他的好朋友 Olivier Danvy 的论文, 讲述如何从各种不同的解释器经过 CPS 变换得出不同种类的抽象机模型。这是我第一次感觉到程序语言的理论对于现实世界的巨大威力, 也让我理解到, 机器并不是计算的本质。机器可以用任何可行的技术实现, 比如集成电路, 激光, 分子, DNA..... 但是无论用什么作为机器的材料, 我们所要表达的语义, 也就是计算的本质, 却是不变的。

而这些还不是我那届 C311 全部的内容。后半学期, 我们开始学习 miniKanren, 一种他自己设计的用于教学的逻辑式语言 (logic programming language)。这个语言类似 Prolog, 但是它把 Prolog 的很多缺点给去掉了, 而且变得更加容易理解。教材是免费送给我们的《The Reasoned Schemer》。在书的最后, 两页纸的篇幅, 就是整个 miniKanren 语言的实现! 我学得比较快, 后来就开始捣鼓这个实现, 把有些部分重新设计了一下, 然后加入了一些我想要的功能。这样的教学, 给了我设计逻辑式语言的能力, 而不只是停留于一个使用者。这是学习 Prolog 不可能做到的事情, 因为 Prolog 的复杂性会让初学者无从下手, 只能停留在使用者的阶段。

我很幸运当初听了他的话去上了这门课, 否则我就不会是今天的我。

独立思考

Dan Friedman 是一个不随波逐流, 有独立思想的人。他的眼里容不下过于复杂的东西, 他喜欢把一个东西简化到容得进几行程序, 把相关的问题理解得非常清楚。他的书是一种独特的“问答式”的结构, 很像孔夫子或者苏格拉底的讲学方式。他的教学方式也非常独特。这在本科生课程 C311 里已经有一些表现, 但是在研究生的课程 B621 里, 才全部的显示出来。

我写过的最满意的一个程序, 自动 CPS 变换, 就是在 C311 产生的。在 C311 的作业里, Friedman 经常加入一些“智力题”(brain teaser), 做出来了可以加分。因为我已经有一定基础, 所以我有精力来做那些智力题。开头那些题还不是很困难, 直到开始学 CPS 的时候, 出现了这么一道: “请写出一个叫 CPSer 的程序, 它的作用是自动的把 Scheme 程序转换成 CPS 形式。”那次作业的其它题目都是要求“手动”把程序变成 CPS 形式, 这道智力题却要求一个“自动”的——用一个程序来转换另一个程序。

我觉得很有意思。如果能写出一个自动的 CPS 转换程序, 那我岂不是可以用它完成所有其它的题目了! 所以我就开始捣鼓这个东西, 最初的想法其实就是“模拟”一个手动转换的过程。然后我发现这真是个怪物, 就那么几十行程序, 不是这里不对劲, 就是那里不对劲。这里按下去一个 bug, 那里又冒出来一个, 从来没见过这么麻烦的东西。我就跟它死磕了, 废寝忘食几乎一星期。经常走进死胡同, 就只有重新开始, 不知道推翻重来多少次。到快要交作业的时候, 我把它给弄出来了。最后我用它生成了所有其它的答案, 产生的 CPS 代码跟手工转换出来的看不出任何区别。当

然我这次我又得了满分（因为每次都做智力题，我的分数总是在100以上）。

作业发下来那天天下课后，我跟 Friedman 一起走向 Lindley Hall (IU 计算机系的楼)。半路上他问我：“这次的 brain teaser 做了没有。”我说：“做了。这是个好东西啊，帮我把其它作业都做出来了。”他有点吃惊，又有点将信将疑的样子：“你确信你做对了？”我说：“不确信它是完全正确，但是转换后的作业程序全都跟手工做的一样。”走向办公室之后，他给了我一篇30多页的论文“Representing control: a study of the CPS transformation”，作者是他的好朋友 Olivier Danvy 和 Andrzej Filinski。然后我才了解到，这是这个方向最厉害的两个。正是这篇论文，解决了这个悬而不决十多年的难题。其实自动的 CPS 转换，可以被用于实现高效的函数式语言编译器。Princeton 大学的著名教授 Andrew Appel 写了一本书叫《Compiling with Continuations》，就是专门讲这个问题的。Amr Sabry（我现在的导师）当年的博士论文就是一个比 CPS 还要简单的变换（叫做 ANF）。凭这个东西，他几乎灭掉了这整个 CPS 领域，并且拿到了终身教授职位。在他的论文发表10年之内也没有 CPS 的论文出现。

Friedman 啊，把这样一个问题作为“智力题”，真有自己的！我开玩笑地对他说：“我保证，我不会把这个程序开源，不然以后你的 C311 学生们就可以拿来作弊了。”回到家，我开始看那篇 Danvy 和 Filinski 的论文。这篇 1991 年的论文的想法，是从 1975 年一篇 Gordon Plotkin 的论文的基础上经过一系列繁琐的推导得出来的，而它最后的结果几乎跟我的程序一模一样，只不过我的程序可以处理更加复杂的 Scheme，而不只是 lambda calculus。我之前完全不知道 Plotkin 的做法，从而完全没有收到他的思想的影响，直接就得到了最好的结果。这是我第一次认识到自己头脑的威力。

第二个学期，当我去上 Friedman 的进阶课程 B621 的时候，他给我们出了同样的题目。两个星期下来，没有其它人真正的做对了。最后他对全班同学说：“现在请王垠给大家讲一下他的做法。你们要听仔细了哦。这个程序价值100美元！”

下面就是我的程序对于 lambda calculus 的缩减版本。我怎么也没想到，这短短 30 行代码耗费了很多人的 10 年的时间才琢磨出来。

```
(define cps
  (lambda (exp)
    (letrec
      ([trivs '(zero? add1 sub1)]
       [id (lambda (v) v)]
       [C~ (lambda (v) `(k ,v))]
       [fv (let ((n -1))
             (lambda ()
               (set! n (+ 1 n))
               (string->symbol (string-append "v" (number->string n)))))]
       [cps1
        (lambda (exp C)
          (pmatch exp
            [x (guard (not (pair? x))) (C x)]
            [(lambda (,x) ,body)
             (C `(lambda (,x k) ,(cps1 body C~)))]
            [(,rator ,rand)
             (cps1 rator
                  (lambda (r)
                    (cps1 rand
                          (lambda (d)
                            (cond
                              [(memq r trivs)
                               (C `(:,r ,d))]
                              [(eq? C C~)
                               ; tail call
                               `(:,r ,d k)]
                              [else
                               (let ([v* (fv)])
                                 `(:,r ,d (lambda (,v*) ,(C v*)))))])))])))]
        (cps1 exp id))))
```

而这还不是 B621 的全部，每一个星期 Friedman 会在黑板上写下一道很难的题目。他不让你看书或者看论文。他有时甚至不告诉你题目里相关概念的名字，或者故意给它们起个新名字，让你想查都查不到。他要求你完全靠自己把这些难题解出来，下一个星期的时候在黑板上给其它同学讲解。他没有明确的评分标准，让你感觉完全没有成绩的压力。

这些题目包括一些很难的问题，比如 church numeral 的前驱 (predecessor)。这个问题，当年是 Stephen Kleene（图灵的学长）花了三个月冥思苦想才做出来的。不幸的是我在 Cornell 就学到了 Kleene 的做法，造成了思维的定势，所以这个训练当时对我来说失去了意义。而我们班上却有一个数学系的同学，出人意料的在一个星期之内做出了一个比 Kleene 还要简单的方法。他的完整的代码（用传统的 lambda calculus 语法表示）如下：

```
λn w z. ((n λl h. h (l w)) (λd.z)) (λx.x)
```

其它的问题包括从 lambda calculus 到 SKI combinator 的编译器，逻辑式（可逆）CPS 变换，实现 Hindley-Milner 类型系统，等等。我发现，就算自认为明白了的东西，经过一番思索，认识居然还可以更进一步。

当然，重新发明东西并不会给我带来论文发表，但是它却给我带来了更重要的东西，这就是独立的思考能力。一旦一个东西被你“想”出来，而不是从别人那里“学”过来，那么你就知道这个想法是如何产生的。这比起直接学会这个想法要有用很多，因为你知道这里面所有的细节和犯过的错误。而最重要的，其实是由此得到的直觉。如果直接去看别人的书或者论文，你就很难得到这种直觉，因为一般人写论文都会把直觉埋藏在一堆符号公式之下，让你看不到背后的真实想法。如果得到了直觉，下一次遇到类似的问题，你就有可能很快的利用已有的直觉来解决新的问题。

而这一切都已经发生在我身上。比如在听说 ANF 之后，我没有看 Amr Sabry 的论文，只把我原来的 CP Ser 程序改了一点点，就得到了 ANF 变换，整个过程只花了十几分钟。而在 R. Kent Dybvig 的编译器课程上，我利用 CPS 变换里面的直觉，改造和合并了 Dybvig 提供的编译器框架的好几个 pass，使得它们变得比原来短小好几倍，却生成更好的代码。

现在我仍然是这样，喜欢故意重新发明一些东西，探索不止一个领域。这让我获得了直觉，不再受别人思想的限制，节省了看论文的时间，而且多了一些乐趣。一个问题，当我相信自己能想得出来，一般都能解决。虽然我经常不把我埋头做出来的东西放在心上，把它们叫做“重新发明”(reinvention)，但是出乎意料的是，最近我发现这里面其实隐藏了一些真正的发明。我准备慢慢把其中一些想法发掘整理出来，发表成论文或者做成产品。

俗话说“给人以鱼，不如授人以渔。”就是这个道理吧。Dan Friedman，谢谢你教会我钓鱼。