我为什么不再做PL人

我为什么不再做PL人

我不做程序语言（PL）的工作已经半年了。在这半年里，我变得快乐了很多，对世界也有了新的观点。现在我想来讲一讲，我为什么不想再做PL的工作和研究。我只希望这些观点可以给正在做PL，或者考虑进入这个领域的人们，作为一份参考。

学校里的PL人

PL看似计算机科学最精髓的部分，事实确实也是这样的。没有任何一个其它领域，可以让你对程序的本质形成如此深入的领悟，然而这并不等于你就应该进入PL的博士班。这是为什么呢？

炒冷饭

PL这个领域几十年来，已经发展到了非常成熟的阶段。这里面的问题，要么在20年前已经被人解决掉了，要么就是类似“停机问题”一样，不可能解决的问题。然而，博士毕业却要求你发表“创新”的论文，那怎么办呢？于是你就只有扯淡，把别人已经解决的问题换个名字，或者制造一些看似新鲜却不管用的概念，在大会上煞有介事的宣讲。俗话说就是“炒冷饭”。

最开头进入这个领域的时候，你可能不觉得是这样，因为似乎有那么多的东西可以学习，那么多的大牛可以瞻仰，那么多的新鲜名词，什么“lambda calculus”啊，“语义”啊，各种各样的“类型系统”啊，这样那样的“逻辑”…… 可是时间久了，看透了，你就发现一些这个圈子里的规律。

崇拜古人

几乎每篇PL领域的论文，里面必有一页弯弯曲曲，让人看花眼的逻辑公式。程序语言的论文，不是用程序来描述，而是用一些老古董的逻辑符号，像这样：

$$(\text{VAR})\ \frac{\Gamma(x) = {}_\wedge T}{\Gamma \vdash^e x : {}_\wedge T} \qquad (\text{SUB})\ \frac{\Gamma \vdash^e E : T \quad T \le T'}{\Gamma \vdash^e E : T'}$$

$$(\text{ABS}_{tp})\ \frac{\Gamma, {}_\wedge\vec{a}, x : {}_\wedge T \vdash^e E : S}{\Gamma \vdash^e \mathbf{fun}[\vec{a}](x:T)E : {}_\wedge(T \xrightarrow{\vec{a}} \diamond S)} \qquad (\text{ABS})\ \frac{\Gamma, {}_\wedge\vec{a}, x : {}_\wedge T \vdash^e E : S \quad \vec{a} \notin \mathrm{tv}(E)}{\Gamma \vdash^e \mathbf{fun}(x)E : {}^\vee(T \xrightarrow{\vec{a}} \diamond S)}$$

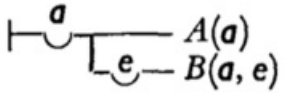
$$(\text{APP}_{tp})\ \frac{\Gamma \vdash^e F : {}^\vee({}_\wedge S \xrightarrow{\vec{a}} {}_\wedge T) \quad \Gamma \vdash^e E : [\vec{R}/\vec{a}]^\vee S}{\Gamma \vdash^e F[\vec{R}](E) : [\vec{R}/\vec{a}]_\wedge T}$$

$$(\text{APP})\ \frac{\begin{array}{c}\Gamma \vdash^e F : {}^\vee({}_\wedge S \xrightarrow{\vec{a}} {}_\wedge T) \quad \Gamma \vdash^e E : S' \quad S' \lhd_{\vec{a}} {}^\vee S \\ S' \le [\vec{R}/\vec{a}]^\vee S \quad [\vec{R}/\vec{a}]_\wedge T \le T' \\ \forall \vec{R}', T''.(S' \le [\vec{R}'/\vec{a}]^\vee S \ \wedge\ [\vec{R}'/\vec{a}]_\wedge T \le T'' \sim T' \ \Rightarrow\ [\vec{R}/\vec{a}]_\wedge T \le [\vec{R}'/\vec{a}]^\vee T)\end{array}}{\Gamma \vdash^e F(E) : T'}$$

$$(\text{SEL})\ \frac{\Gamma \vdash^e E : {}^\vee\{x : \diamond T\}}{\Gamma \vdash^e E.x : T} \qquad (\text{REC})\ \frac{\Gamma \vdash^e E_1 : \diamond T_1 \quad \ldots \quad \Gamma \vdash^e E_n : \diamond T_n}{\Gamma \vdash^e \{x_1 = E_1, \ldots, x_n = E_n\} : {}_\wedge\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\}}$$

绝大部分PL领域的专家们，似乎都酷爱逻辑符号，视逻辑学家高人一等。这种崇尚古人的倾向，使得PL专家们看不见这些符号背后，类似电路一样的直觉。他们看不见逻辑学的历史局限，所以他们也许能够发展和扩充一个理论，却无法创造一个新的。

说到古人，却并不是所有古人都这么晦涩。如果你考古一下就会发现，其实现代逻辑学的鼻祖Gottlob Frege最初的论文里，是没有这些稀奇古怪的符号的。他整篇论文都在画图，一些像电路一样的东西。比如下图，就是Frege的创始论文《Begriffsschrift》里最复杂的“公式”之一：



你可以把这里的每根线理解成一根电线。图1里那些诡异的逻辑符号，都是一些好事的后人（比如Gentzen）加进去的，最后搞得乌七八糟，失去了Frege理论的简单性。所以PL专家们虽然崇尚古人，却没有发现大部分古人，其实并没能获得鼻祖Frege的真传。

如果你看透了那些公式，自己动手实现过各种解释器，就会发现PL论文里的那些公式，其实相当于解释器的代码，只不过是用一种叫做“XX逻辑”的晦涩的语言写出来的。逻辑，其实本质上是一种相当落伍的程序语言。如果你精通解释器的代码，也许就会发现，这些公式其实用非常蹩脚的方式，实现了哈希表等数据结构。逻辑语言只运行于逻辑学家的脑子里面，用它写出的代码一样可能有bug，而且由于这语言如此障眼难读，而且没有debugger，所以bug非常难发现。逻辑学家们成天为自己的设计失误和bug伤透了脑筋，PL专家们却认为他们具有数学的美感，是比自己聪明的高人 :)

所以当你看透了所有这些，就会发现PL的学术界，其实反反复复在解决一些早已经解决了的问题，只不过给它们起了不同的名字，使用不同的方式来描述。有时候好几个子领域，其实解决的是同一个问题，然而每个子领域的人，却都说自己的问题在本质上是不一样的，号称自己是那个子领域的鼻祖。甚至有人在20多年的时间里，制造出一代又一代的PhD和教授职位。他们的理论一代代的更新，最后却无法解决实际的问题。所谓的“控制流分析”（control-flow analysis，CFA），就是这样的一个子领域。

不知道谁是真的高人

进入一个领域做研究，你总该知道那些人是真正厉害的。可惜的是，PL这个领域里，你往往不知道谁是真正掌握了精髓的学者，甚至好几年之后你仍然蒙在鼓里。我的历史教训是，写教科书的人，往往不是最聪明，最理解本质的。真正深刻的PL研究者，你可能根本没听说过他们的名字。

一般程序员提到PL，就会跟“编译器”这个领域混淆在一起，就会想起大学时候上编译器课，看《龙书》时焦头烂额的情景。然后由于斯德哥尔摩综合症，他们就会崇拜龙书的作者们。直到遇到了真正厉害的PL专家，你才发现编译器这个领域，跟PL根本是两回事，它其实比PL要低一个档次，里面充满了死记硬背的知识甚至误导。龙书的作者，其实也不是最厉害的编译器作者，他们更不是合格的PL专家。

上过“正统”的PL课程的学生，往往用一本经典大部头教材叫《TAPL》，然后就会误认为此书的作者是最厉害的PL专家，然而他们再一次被名气给蒙蔽了。TAPL这书其实不但照本宣科，没有揭示实质，而且冗长没有选择，有用的没用的过时的理论，一股脑的灌输给你。等你研究到了所谓“交集类型”（intersection types），看到TAPL作者当年的博士论文才发现，其实他把简单的问题搞复杂了，而且那些理论几乎完全不能实用。真正厉害的intersection types专家，其实默默无闻的待在Boston University，而且研究到最后，intersection types这个领域其实被他们证明为完全不能实用。

由于TAPL这本书，以及ML，Haskell等语言在PL界的“白象”地位，于是很多人又对Hindley-Milner类型系统（HM）充满了崇敬之情，以为HM系统的发明者Robin Milner是最厉害的PL学者。他的确不错，然而等你随手就能实现出HM系统，看清了它的实质，就会发现所有这样能够“倒推”出类型的系统，其实都具有很大的局限性。

HM系统的“unification”机制，依赖于数学上的“等价关系”，所以它不可能兼容子类型（subtyping）关系。原因很简单：因为子类型没有交换性，不是一个等价关系。而子类型关系却是对现实世界进行直观的建模所必不可少的，于是你就发现Haskell这类基于HM系统的语言，为了弥补这些缺陷而出现各种“扩展”，却永远无法达到简单和直观。一开头就错了，所以无论Haskell如何发展，这个缺陷也无法弥补。如果没有了HM系统，Haskell就不再是Haskell。

Robin Milner的另外一个贡献π-calculus，虽然看起来吓人，其实看透了之后你发现它里面并没有很多东西。π-calculus对并发进行“建模”，却不能解决并发所带来的各种问题，比如竞争（race condition）。实际上普通的语言也能对并发进行简单的建模，所以π-calculus其实只停留于纸面上，不可能应用到现实中去。跟π-calculus类似的一个概念CSP也有类似的问题，属于“白象理论”。很多语言（比如Go）扯着CSP的旗号，引起很多人无厘头的膜拜，可见白象的威力有多大 :)

我在学校研究PL的时候就是这样，每天都发现天外有天，每天都发现曾经的偶像其实很多时候是错觉。最后我发现，PL领域其实最后就剩下那么一点点实质的内容，其它的都是人们造出来的浮云。所以每当有人问我推荐PL书籍，我都比较无语，因为我的PL知识只有非常少数是看书得来的。自己动手琢磨出来的知识，才是最管用的。

没人知道你是谁

PL的学生还有一个问题，那就是毕业后工作不好找。只有极少数公司（像微软，Intel，Oracle）里的少数团队，可以发挥PL专家的特殊才能。绝大部分其它公司根本不知道PL是什么，PL专家是干什么的。你跟他们说你的专业是“程序语言”，他们还以为你只是学会了“编程”而已，还问你想做“前端”还是“后端” :) 诚然，PL学生一般都有很好的编程能力，然而公司往往只关心自己的实际需求。PL学生毕业之后，很容易被普通公司作为没有任何专长的人对待。

另外，PL的圈子相当的小，而且门派宗教观念严重，所以就算你从名师手下毕业，想进入另一个老师的门徒掌权的公司，很可能因为两个门派的敌视而无法被接纳，就算进去了也经常会因为对于PL的理念不同而发生冲突。所以，学习PL最精髓的理论是有好处的，然而进入PhD投身PL的研究，我觉得应该三思。

公司里的PL人：过度工程

PL人在学校里跟着教授炒冷饭，毕业进入了公司之后，他们的行为方式还是非常类似。他们喜欢在公司里做的一件事情，叫做“过度工程”。本来很直接，很容易解决的一个问题，非要给你扯到各种炫酷的PL名词，然后用无比复杂的方案来解决。

有一些PL人喜欢推广他们认为高大上的语言，比如Haskell，OCaml，Scala等。这些语言在PL学术界很受尊重，所以他们以为这些语言能够奇迹般的解决实际的问题，然而事实却不是这样的。事实是，这些学术界出来的语言，其实缺乏处理现实问题的机制。为了能够在学术上证明程序的所谓“正确性”，而且由于类型系统本身的局限性，这些语言往往被设计得过于简单，具有过度的约束性，以至于表达能力欠缺。

最后，你发现用这些语言来写代码，总是这也不能做，那也不能做，因为你要是那么做了，编译器就无法发现“类型错误”。到最后你发现，这些语言的约束，其实是无需有的。如果放宽这些约束，其实可以更优雅，更简单的对问题进行建模。对正确性的过分关注，其实导致了PL人选择蹩脚的语言，写出绕着弯子，难以理解的代码。

还有一类PL人，喜欢设计不必要存在的语言。因为他们认为设计语言是PL人的特异功能，所以随时随地都想把问题往“语言设计”的方向上靠。这样的趋势是非常危险的，因为有原则的PL人，其实都明白一条重要的道理：不到万不得已的时候，千万不要制造语言。

很多PL人在公司里盲目的制造新的语言，导致的问题是，到最后谁也无法理解这种新语言写出来的代码。这一方面是新语言必然导致的结果，另一方面是由于，并不是每一个PL人都有全面的知识和很好的“品味”。每个PL学生毕业，往往只深入研究了PL的某个子领域，而对其它方面只是浮光掠影，所以他们有可能在那上面犯错。有些PL人喜欢照猫画虎，所以可能盲目的模仿Go语言，Haskell或者Python的特性，设计出非常蹊跷难用的语法。这些新的语言，其实让其他人苦不堪言。最后你发现，他们声称新语言能解决的问题，其实用像Java一样的老语言，照样可以很容易的解决。

喜欢钻牛角尖，把问题搞复杂，就是很多公司里的PL人的共同点。制造语言是PL人应该尽量避免的事情，这恰恰跟PL人的专长是矛盾的。所以有原则的PL人，生活怎么可能不苦 :)

PL人的天才病

很多研究 PL 的人喜欢看低其它程序员，认为自己能设计实现程序语言，就是天之骄子。我之所以从 Dan Friedman 那里学到了好东西，却没有成为他的 PhD 学生，一方面就是因为看不惯围绕在他身边那些自认为是“天才”的人。

总是有那么一群本科生，自认为掌握了 Friedman 所讲授的精髓，所以高人一等。于是我就经常无奈的看着他们，吵吵闹闹的宣讲他们解决的“新问题”，貌似什么了不起的发明一样，受到 Friedman 的肯定就受宠若惊的样子。而其实呢，那些都是我几年前就已经试过并且抛弃的方案……

其它的 PL 人，包括 PhD 学生，也有一样的毛病。不管在三流大学，还是在 Harvard，Princeton，MIT 这样的“牛校”出来的，只要是 PL 人，几乎必然有这种天才作风。另外你可能不知道的是，牛校往往并不产出优秀的 PL 人才。像 Stanford，Berkeley，MIT 这样的传统 CS 牛校，其实在 PL 方面是相当差的。

这种天才病的危害在于，它蒙蔽了这些人的眼睛。他们不再能设计出让“普通人”可以容易使用的产品。如果你不会用，他们就会嘲笑你笨，而其实呢，是因为他们的设计不好。他们喜欢用含混晦涩的方式（所谓“函数式”）的写法来构造代码，让其它人阅读和修改都极其困难，……

这些所谓天才，看不到简单直观的解决方案，为了显示自己的聪明而采用繁复的抽象，其实是一种愚蠢。真正的天才，必须能够让事情变得简单。