Python: The Full Monty, A Tested Semantics for the Python Programming Language

大家好，今天和大家分享的论文是 Python: The Full Monty，然后它有一个副标题 A Tested Semantics for the Python Programming Language。首先 the full monty 是一个不太常见的英文俗语吧[1]，有 all that you need 的意思，然后结合副标题我们可以知道这篇文章是在做 Python 语言的语义建模，大家如果了解这类工作的话就会知道，由于完整建模一个通用编程语言是比较困难的，所以这类工作在命名的时候常常会使用 light 或 lite、featherweight 这样的词汇，那么本文通过 the full monty 想要表达的是希望给出一个比较完整的语义，并且检查语义模型的正确性，就是这里所谓的是一个 tested、经过测试的 semantics。

这篇文章发表在 OOPSLA 2013，OOPSLA 是程序语言方向四个重要会议之一，主要关注面向对象的编程系统、语言和应用。本文的作者包括 Joe 等 8 个人，其中 Daniel Patterson[5] 上学期也介绍过他后来围绕语言互操作语义的工作。

研究动机

首先本文给了一个例子说明为什么需要形式化的语义，因为语义可以精确地分析程序或证明程序的性质，本文用一个反例说明很多 Python 程序分析工具是不可靠的，比如多个 IDE 的变量重命名重构功能，即用户选中一个变量，对其重命名重构应该改变这个变量在所有定义和使用点的名字。这个例子是关于 Python 的作用域，Python 允许在函数内定义类，这里外部函数的形参和内部类的变量有相同的名字 x，那么在这个类方法中 x 对应函数形参，C.x 对应类变量。

```
def f(x):
  class C:
    x = "C's x"
    def meth(self):
      return x + ', ' + C.x
  return C
f('input x')().meth()
```

可是在 PyCharm 和 PyDev 中，重命名重构函数形参 x 却改变了类变量，导致得到一个错误的程序。在 PyCharm 中重命名重构类变量的使用 C.x 得到的也是错误的结果。在 PyDev 中重命名重构类变量的定义几乎做对了，只是它连字符串中的 x 都改掉了。

语义规定了程序的含义，或者说如何确定程序的值。比如我们知道 Python 有一个灵活的语法，这给分析或证明带来了困难，但是其实很多只是语法糖，就比如其实类型定义和 type 函数调用就是等价的。

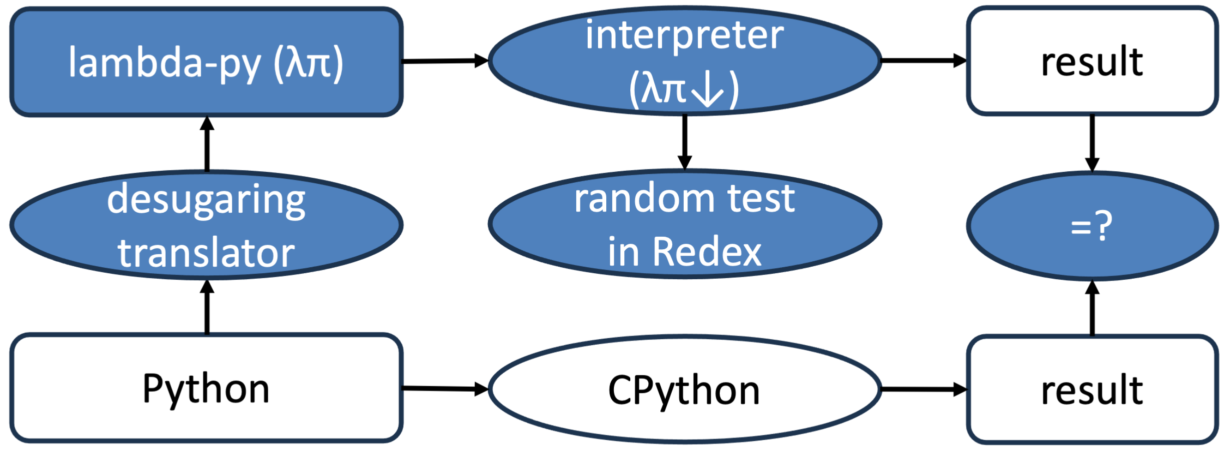
```
class X:
  a = 1

X = type('X', (object,), dict(a=1))
```

主要贡献

由此，本文的主要贡献就是把 Python 的语义表示为一个核心语言 lambda-py 及其规约规则，基于 lambda-py 做分析或证明会比基于 Python 要更容易，因为它更简单，同时也更精确可靠，因为它是形式化的，并且有机械实现，在本文中还是经过测试的。

本文基于 Redex 框架[6]实现 lambda-py 语言及其规约语义，通过一个去糖化翻译的过程把 Python 编译到 lambda-py，通过一个解释器执行 lambda-py 程序，这个解释器相比于 CPython 可以看做一个 Python 实现，将两者的结果对比可以说明语义模型的正确性，同时本文通过 CPython 测试套检查解释器执行语义的正确性。



方法思路（语义）

首先给出 lambda-py 的值和表达式。值是一个对象或一个引用，其中对象是一个三元组，第一元代表对象的类型，第二元代表对象的内容，第三元是一个字典，记录对象的域。其中对象的内容是一个原始值，如数值、列表、函数等。堆由值的指针组成，当指向的位置未初始化时，对应的未定义值用这个骷髅头记号表示。

```
heap         Σ ::= ((ref v+undef) ...)
           ref ::= natural
value   v, val ::= ⟨val, mval, {string: ref, ...}⟩        object <class, content, {fields}>
                 | ⟨x, mval, {string: ref, ...}⟩
                 | @ref | (sym string)                    reference
       v+undef ::= v | ☠
       e+undef ::= e | ☠                                  uninitialized heap location
             t ::= global | local
meta-value mval ::= (no-meta) | number | string | meta-none
                 | [val ...] | (val ...) | {val ...}
                 | (meta-class x) | (meta-code (x ...) x e)
                 | λ(x ...) opt-var.e
       opt-var ::= (x) | (no-var)
expression   e ::= v | ref | (fetch e) | (set! e e) | (alloc e)
                 | e[e] | e[e := e]
                 | if e e e | e e
                 | let x = e+undef in e
                 | x | e := e | (delete e)
                 | e (e ...) | e (e ...)*e | (frame e) | (return e)
                 | (while e e e) | (loop e e) | break | continue
                 | (builtin-prim op (e ...))
                 | fun (x ...) opt-var e
                 | ⟨e, mval⟩ | list⟨e, [e ...]⟩
                 | tuple⟨e, (e ...)⟩ | set⟨e, (e ...)⟩
                 | (tryexcept e x e e) | (tryfinally e e)
                 | (raise e) | (err val)
                 | (module e e) | (construct-module e)
                 | (in-module e ε)
```

在此基础上，小步语义可以用这样的规约关系表示，即求值产生的不可再分的状态变化，状态是一个三元组，包含表达式、全局环境和堆。

$$(e\ \varepsilon\ \Sigma) \longrightarrow (e\ \varepsilon\ \Sigma)$$

举例来说，在求值上下文 E 中有这样一个列表创建表达式，对它求值首先会改变堆，因为需要分配一个新的对象，这个对象的类型为 val_c，原始值是一个列表，列表对象没有域。分配返回一个引用，它指向堆上新分配的对象。

$$(E[\ \texttt{list}\langle val_c, [val\ \ldots]\rangle]\ \varepsilon\ \Sigma) \qquad \text{[E-List]}$$

$$\longrightarrow (E[@ref_{new}\ ]\ \varepsilon\ \Sigma_1)$$

$$\texttt{where}\ (\Sigma_1\ ref_{new}) = \texttt{alloc}(\Sigma, \langle val_c, [val\ \ldots], \{\}\rangle)$$

这里额外解释一下，引入求值上下文是为了表示求值顺序，环境则是一个变量到地址的映射表。类似地，我们可以给出 lambda-py 中所有表达式的规约关系。本文有选择地介绍了一些特别的语言特性及其语义建模。

首先是一阶函数，在 Python 里，函数也是一个对象，特殊地，Python 函数是一个可调用的对象，并带有一个可变域。比如说，Python 支持这样的写法：

```
def f():
  return f.x
```

```
f.x = -1
f()
```

可以看到，在函数定义时用到了未定义的变量，为了支持这一特性，在 lambda-py 中函数对象的原始值是这样的。

$$\lambda(x\ \ldots)\ \textit{opt-var}.\,e$$

和一般的 lambda 表达式对比，这里多了一个可选值，它是一个额外的参数元组，当函数调用使用了函数变量列表中未定义的对象，就会进行分配并绑定到可选参数元组。

这里会涉及到对象的域操作，对应的规约规则是：

$$\begin{aligned}
&(E[@ref_{obj}\ [@ref_{str}\ := val_1]]\ \varepsilon\ \Sigma) \qquad \text{[E-SetFieldUpdate]}\\
\longrightarrow\ &(E[val_1]\ \varepsilon\ \Sigma[ref_1{:=}val_1])\\
&\text{where } \langle any_{cls1}, mval, \{string_2{:}ref_2, \ldots, string_1{:}ref_1, string_3{:}ref_3, \ldots\}\rangle = \Sigma(ref_{obj}),\\
&\qquad \langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})
\end{aligned}$$

$$\begin{aligned}
&(E[@ref_{obj}\ [@ref_{str}\ := val_1]]\ \varepsilon\ \Sigma) \qquad \text{[E-SetFieldAdd]}\\
\longrightarrow\ &(E[val_1]\ \varepsilon\ \Sigma_2)\\
&\text{where } \langle any_{cls1}, mval, \{string{:}ref, \ldots\}\rangle = \Sigma(ref_{obj}),\\
&\qquad (\Sigma_1\ ref_{new}) = \mathrm{alloc}(\Sigma, val_1),\\
&\qquad \langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str}),\\
&\qquad \Sigma_2 = \Sigma_1[ref_{obj}{:=}\langle any_{cls1}, mval, \{string_1{:}ref_{new}, string{:}ref, \ldots\}\rangle],\\
&\qquad (\mathrm{not}\ (\mathrm{member}\ string_1\ (string\ \ldots)))
\end{aligned}$$

$$\begin{aligned}
&(E[@ref\ [@ref_{str}\ ]]\ \varepsilon\ \Sigma) \qquad \text{[E-GetField]}\\
\longrightarrow\ &(E[\Sigma(ref_1)]\ \varepsilon\ \Sigma)\\
&\text{where } \langle any_{cls1}, string_1, any_{dict}\rangle = \Sigma(ref_{str}),\\
&\qquad \langle any_{cls2}, mval, \{string_2{:}ref_2, \ldots, string_1{:}ref_1, string_3{:}ref_3, \ldots\}\rangle = \Sigma(ref)
\end{aligned}$$

我们挑一个看起来复杂一点的新增域看一下，对于表达式 e[e=v]，首先用原对象的指针从堆上取出原对象，它的类型是 cls1，值是 mval，有这些域。首先我们需要分配新的域的值 val1，refnew 指向它，这个域的名字是字符串 string1，然后我们更新原对象，类型和原始值不变，域字典增加一个 string1 到 refnew 的映射，域新增的条件是 string1 不在原来域字典的键中。

结合魔术域的特性，就形成了 Python 特有的动态性和灵活性。例如只要一个对象带有 `__add__` 域，那么它就可以作为加数；只要一个对象带有 `__call__` 域，它就可以被调用。甚至取一个对象的域也是一个 `__getattribute__` 域方法调用。回到前面，一个对象，它本身不是函数或类等内置的可执行的对象，但只要实现它的 `__call__` 属性，那么它就可以被调用了。通过核心语义和去糖化，这些复杂特性的分析就变得可行了。

Python 还有一些更加复杂的特性，比如作用域。Python 混合了不同的作用域概念，函数参数和赋值语句的左手侧使用的是词法作用域，使用 let 表达式去糖化、使用替换求值就可以常规地处理。

一个动态性的问题是如果变量绑定出现在条件分支中呢？比如下面的程序。

```
def f(y):
  if y > .5:
```

```
    x = "big"
  else:
    pass
  return x
f(0) # UnboundLocalError
f(1) # "big"
```

解决方法是在去糖化的过程中增加两步，首先收集前面词法作用域的变量，然后对于所有的这些局部变量，把函数包裹到局部变量赋初值为未定义的 let 绑定中。即上述程序去糖化之后形如：

```
let  f = ☠  in
 f :=
 fun (y) (no-var)
  let  x = ☠  in
   if (builtin-prim “num>” (y ⟨%float,0.5⟩))
      x := ⟨%str,“big”⟩
      none
  (return x)

 f (⟨%int,0⟩)
 f (⟨%int,1⟩)
```

然后结合查找未初始化变量的报错规则即可。

```
   (E[ref] ε Σ)                                        [E-GetVarUndef]
⟶ (E[(raise ⟨%str,“Uninitialized local”,{}⟩)] ε Σ)
    where Σ = ((ref₁ v+undef₁) ... (ref ☠) (refₙ v+undefₙ) ...)
```

这里其实已经展现了一个分析这种动态性导致的程序错误的方法，可以看到语义模型给复杂分析带来的好处。

进一步地，我们看看 Python 中闭包的设计。首先一般地，内部作用域可以看到外部作用域，比如这个程序内部函数 g 使用的变量 x 在外部函数 f 中定义。

```
def f():
  x = "closed-over"
  def g():
    return x
```

```
    return g
  f()() # ==> "closed-over"
```

但是，封闭变量 x 对于内部函数而言是不可变的，赋值语句总是新分配一个变量，此时 x 作为内部作用域中赋值语句的左手侧，它可以和外部的 x 同名但不是一个变量。

```
def g():
  x = "not affected"
  def h():
    x = "inner x"
    return x
  return (h(), x)
g() # ==> ("inner x", "not affected")
```

为了支持在内部作用域改变封闭变量，Python 3 引入了 nonlocal 关键字。

```
def g():
  x = "not affected"
  def h():
    nonlocal x
    x = "inner x"
    return x
  return (h(), x)
g() # ==> ("inner x", "inner x")
```

对应地，前面的去糖化翻译需要增加一步，就是只对非 nonlocal 的词法作用域变量做未定义绑定包裹。

更进一步地，Python 还支持函数定义嵌套类定义，以及一部分动态作用域。这里不再展开，前者的处理思路和闭包类似，需要设计更复杂的去糖化翻译，见论文第 4.2.3 和第 4.2.4 节。后者需要操作全局环境，就是前面的 ε，见附录第 3 节。

实现与测试

本文在实现上遵循两个目标：

1. 完整性：去糖化能把所有 Python 程序翻译到 lambda-py
2. 一致性：lambda-py 程序求值和翻译前的 Python 程序求值有相同的值

由于 Python 本身没有形式化规范，所以一致性无法给出证明。所以本文通过测试实现以上两个目标。

本文去糖化翻译的过程如下：

1. 处理类中的定义
2. 处理变量绑定，包括 nonlocal 和 global 变量的处理，此时程序可以视作一个纯词法作用域的中间表示
3. 处理类上的操作，如前面魔术域等所示，类和类操作都会变成函数调用，其他处理如 for 循环去糖化等

4. 处理生成器，这部分比较复杂，需要一些预备知识，可以看论文第 4.1 和第 4.3 节

语义执行的过程如下：

1. 解析和去糖化依赖的 Python 实现的库
2. 解析和去糖化目标 Python 程序
3. 直接构建无法处理的内置库的语法树
4. 组合前面 3 部分得到目标程序对应的 lambda-py 表达式
5. lambda-py

由于许多高级特性只是语法糖，对语义测试没有帮助，因此本文测试了 CPython 测试套中的 205 个文件，结果全部相同。没有使用 Python unittest 单元测试来进行测试的原因是 lambda-py 不支持 unittest 使用的反射、外部库等特性。

此外，本文基于 Redex 框架实现语义模型，Redex 支持根据规约规则随机生成项来直接测试语义[7]，有很好的覆盖率。本文选择间接测试解释器的原因是性能更好，并且部分环境和堆需要手工构建阻碍了直接测试可执行语义。

优缺点与启发

我觉得本文是语义工程一个基础而实用的参考论文。首先它的叙述很清晰，和同类的工作比如此前我们提到的 Daniel Patterson 的后续工作相比，可能它没有那么困难，例如没有复杂的证明，也没有引入一些复杂的程序构造。但是它在简单的同时给出了实际且可发表的贡献，比如选择哪些部分来形式化，选择哪些特性来介绍，对比论文和附录可以看到这种选择。

缺点我觉得这篇文章可能在 POPL 或 PLDI 等会议的角度来看会比较缺乏创新，在软工的文章来看会缺少评估和应用，OOPSLA 的评委同样可能会有这些疑问。此外，在细节上，动机这个例子我觉得并不好，一方面它和后文有些脱节，一方面不是有了语义就可以做可靠的分析，尤其本文的正确性是通过测试而且是不太完整的测试说明的，而且 VSCode 对于这个例子的重命名重构是正确的，是因为它的 Python 插件是可靠的、基于语义的吗，可能也不是。

我读这篇文章的原因一方面是因为刚才说的优点，我觉得它是一个语义工程入手的很好的参考，先构建一个基础，说清它的贡献，以后可以再往里面加入理论上更新颖的内容，或是应用到某一个具体问题。另一方面是 Python 语义的参考，我正在做这样一个工作。刚才也提到了本文不支持外部库，但是不仅标准库、测试套中包含大量外部库，第三方包很多也是多语言的。我希望定义一个 Python/C 的多语言的语义，核心语言选择的是字节码形式加 Python/C API，选择字节码形式一方面符合基于 Python/C API 的互操作先编译后加载和链接的机制，一方面可以复用语言本身的编译工具链，省去了这里去糖化翻译器的实现和正确性证明。这个语义的执行过程类似于一个 VM，后续也可以用于探索更多多语言的语义特性和分析证明。

参考

[1] the full monty, Cambrige dictionary, https://dictionary.cambridge.org/dictionary/english/full-monty

[2] JNI Light: An Operational Model for the Core JNI, Gang Tan, APLAS 2010, https://link.springer.com/chapter/10.1007/978-3-642-17164-2_9

[3] X86lite documentation, Steve Zdancevic, https://www.cs.princeton.edu/courses/archive/spring19/cos320/hw/x86lite.shtml

[4] Featherweight Java: A Minimal Core Calculus for Java and GJ, Atsushi Igarashi and Benjamin C Pierce and Philip Wadler, TOPLAS 2001, https://dl.acm.org/doi/abs/10.1145/503502.503505

[5] Daniel Patterson, https://dbp.io/

[CS173] CS 173: Semantics Engineering, Shriram Krishnamurthi, https://cs.brown.edu/courses/cs173/2012/

[6] Semantics Engineering with PLT Redex, Matthias Felleisen and Robert Bruce Findler and Matthew Flatt, https://mitpress.mit.edu/9780262062756/semantics-engineering-with-plt-redex/

[7] Randomized testing in PLT Redex, Casey Klein and Robert Bruce Findler, http://users.cs.northwestern.edu/~robby/pubs/papers/scheme2009-kf.pdf