Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

June 6, 2019

http://github.com/ChristianoBraga/PiFramework

Compiler pipeline

$\begin{array}{l} \text{source} \\ \text{code} \end{array} \xrightarrow{\text{lexer}} \text{tokens} \xrightarrow{\text{parser}} \begin{array}{l} \text{concrete} \\ \text{syntax} \\ \text{tree} \end{array} \xrightarrow{\text{AST transformer}} \begin{array}{l} \text{abstract} \\ \text{syntax} \\ \text{tree} \end{array} \xrightarrow{\text{type checker}} \begin{array}{l} \text{abstract} \\ \text{syntax} \\ \text{tree} \end{array} \xrightarrow{\text{code generator}} \begin{array}{l} \text{machine} \\ \text{code} \end{array} \xrightarrow{\text{optimizer}} \begin{array}{l} \text{optimized} \\ \text{machine} \\ \text{code} \end{array}$

Instituto de
Computação
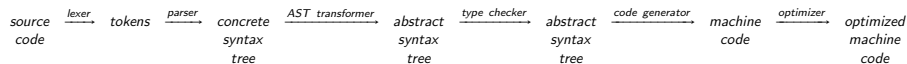
Compiler pipeline and formal languages

$$\text{source} \atop \text{code} \xrightarrow[\text{lexer}]{\text{Regular Grammar}} \text{tokens} \xrightarrow[\text{parser}]{\text{ContextFree Grammar}} \text{concrete syntax tree} \xrightarrow[\text{AST transformer}]{\text{ContextFree Grammar}} \text{abstract syntax tree} \xrightarrow[\text{type checker}]{\text{ContextSensitive Grammar}} \text{abstract syntax tree} \xrightarrow[\text{code generator}]{\text{Turing Machine}} \text{machine code} \xrightarrow[\text{optimizer}]{\text{Turing Machine}} \text{optimized machine code}$$
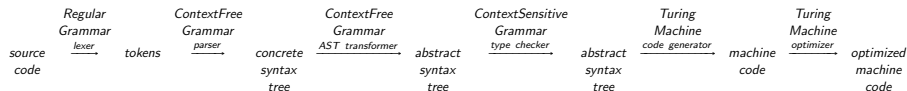
Compiler pipeline with the π Framework
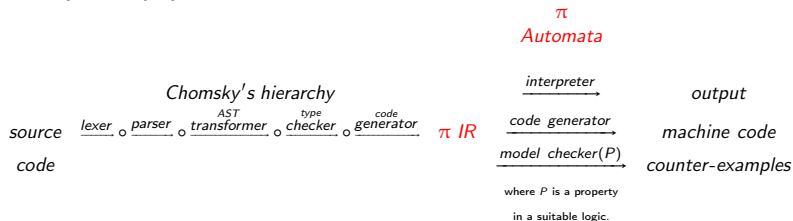
- π IR defines a set of constructions common to many programming languages.
- π IR constructions have a formal automata-based semantics in π automata.
- One may execute (or validate) a program in a given language by running its associated π IR program.
- π Framework:
  http://github.com/ChristianoBraga/PiFramework
- Notes on Formal Compiler Construction with the π Framework:
  https://github.com/ChristianoBraga/PiFramework/blob/master/notes/notes.pdf.

A calculator

We wish to compute simple arithmetic expressions such as $5 * (3 + 2)$.

A calculator: Lexer

$\langle digit \rangle$ $::= [0..9]$

$\langle digits \rangle$ $::= \langle digit \rangle^+$

$\langle boolean \rangle$ $::=$ 'true' | 'false'

A calculator: concrete syntax

$\langle exp \rangle$ ::= $\langle aexp \rangle$ | $\langle bexp \rangle$

$\langle aexp \rangle$ ::= $\langle aexp \rangle$ '+' $\langle term \rangle$ | $\langle aexp \rangle$ '-' $\langle term \rangle$ | $\langle term \rangle$

$\langle term \rangle$ ::= $\langle term \rangle$ '*' $\langle factor \rangle$ | $\langle term \rangle$ '/' $\langle factor \rangle$ | $\langle factor \rangle$

$\langle factor \rangle$ ::= '(' $\langle aexp \rangle$ ')' | $\langle digits \rangle$

$\langle bexp \rangle$ ::= $\langle boolean \rangle$ | '~' $\langle bexp \rangle$ | $\langle bexp \rangle$ $\langle boolop \rangle$ $\langle bexp \rangle$
                 | $\langle aexp \rangle$ $\langle iop \rangle$ $\langle aexp \rangle$

$\langle boolop \rangle$ ::= '=' | '/\\' | '\\/'

$\langle iop \rangle$ ::= '<' | '>' | '<=' | '>='

A calculator: abstract syntax

⟨*exp*⟩   ::= ⟨*digits*⟩ | ⟨*boolean*⟩ | ⟨*exp*⟩ ⟨*bop*⟩ ⟨*exp*⟩

⟨*bop*⟩   ::= '+' | '−' | '*' | '/' | '=' | '\/' | '/\' | '<' | '>' | '<=' | '>='

A calculator: π denotations I

Let D in <digits>, B in <boolean> and E_1, E_2 in <exp>,

$$[\![D]\!]_\pi = Num(D) \tag{1}$$

$$[\![B]\!]_\pi = Boo(B) \tag{2}$$

$$[\![E_1 + E_2]\!]_\pi = Sum([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{3}$$

$$[\![E_1 - E_2]\!]_\pi = Sub([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{4}$$

$$[\![E_1 * E_2]\!]_\pi = Mul([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{5}$$

$$[\![E_1 / E_2]\!]_\pi = Div([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{6}$$

$$[\![E_1 < E_2]\!]_\pi = Lt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{7}$$

$$[\![E_1 <= E_2]\!]_\pi = Le([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{8}$$

$$[\![E_1 > E_2]\!]_\pi = Gt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{9}$$

$$[\![E_1 >= E_2]\!]_\pi = Ge([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{10}$$

A calculator: π denotations II

- π denotations are *functions* $\llbracket \cdot \rrbracket_\pi : AST \to \pi\ IR$, where *AST* denotes the *datatype* for the abstract syntax tree and $\pi\ IR$ denotes the datatype for $\pi\ IR$ programs.

- Note that $\llbracket \cdot \rrbracket_\pi$ has *trees* as parameters, instances of *AST*. The example expression $5 * (3 + 2)$ becomes

A calculator: π denotations III
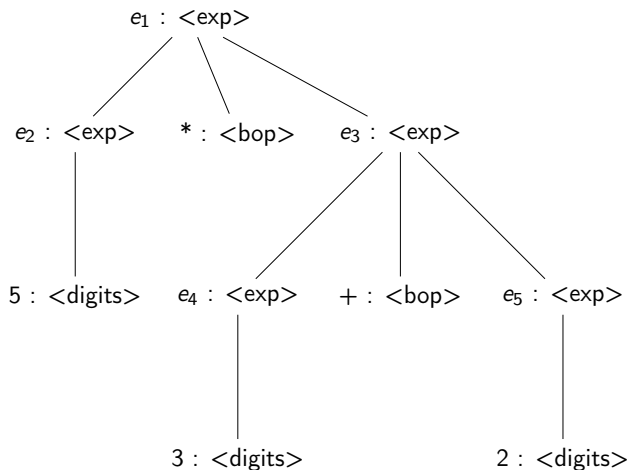
A calculator: π denotations IV

$$\llbracket 5 * (3+2) \rrbracket_\pi = Mul(\llbracket 5 \rrbracket_\pi, \llbracket (3+2) \rrbracket_\pi) \qquad \text{by Equation 5}$$

$$Mul(\llbracket 5 \rrbracket_\pi, \llbracket (3+2) \rrbracket_\pi) = Mul(Num(5), \llbracket (3+2) \rrbracket_\pi) \qquad \text{by Equations 1}$$

$$Mul(num(5), \llbracket (3+2) \rrbracket_\pi) = Mul(Num(5), Sum(\llbracket 3 \rrbracket_\pi, \llbracket 2 \rrbracket_\pi)) \qquad \text{by Equation 3}$$

$$Mul(Num(5), Sum(\llbracket 3 \rrbracket_\pi, \llbracket 2 \rrbracket_\pi)) = Mul(Num(5), Sum(Num(3), \llbracket 2 \rrbracket_\pi)) \qquad \text{by Equation 1}$$

$$Mul(Num(5), Sum(Num(3), \llbracket 2 \rrbracket_\pi)) = Mul(Num(5), Sum(Num(3), Num(2))) \qquad \text{by Equation 1}$$

A calculator: executing π IR with π automata

A π automaton is a 5-tuple $\mathscr{A} = (G, Q, \delta, q_0, F)$, where G is a context-free grammar, Q is the set of states, q_0 is the initial state, $F \subseteq Q$ is the set of final states and

$$\delta : L(G)^* \times L(G)^* \times Store \to Q,$$

where $L(G)$ is the language generated by G and $Store$ represents the memory. (Elements in a set S^* are represented by terms $[s_1, s_2, \ldots, s_n]$.)

$$\delta([Mul(Num(5), Sum(Num(3), Num(2))], \varnothing, \varnothing) = \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \varnothing, \varnothing)$$

$$\delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \varnothing, \varnothing) = \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \varnothing)$$

$$\delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \varnothing) = \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \varnothing)$$

$$\delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \varnothing) = \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \varnothing)$$

$$\delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \varnothing) = \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \varnothing)$$

$$\delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \varnothing) = \delta([\#MUL], [Num(5), Num(5)], \varnothing)$$

$$\delta([\#MUL], [Num(5), Num(5)], \varnothing) = \delta(\varnothing, [Num(25)], \varnothing)$$

$$\delta(\varnothing, [Num(25)], \varnothing) = Num(25)$$

Excerpt of π IR expressions

$\langle Statement \rangle$ ::= <Exp>

$\langle Exp \rangle$ ::= <ArithExp> | <BoolExp> <Exp>

$\langle ArithExp \rangle$ ::= 'Num'(<digits>) | 'Sum'(<Exp> , <Exp>) |
'Sub'(<Exp>, <Exp>) | 'Mul'(<Exp>, <Exp>)

$\langle BoolExp \rangle$ ::= 'Eq'(<Exp>, <Exp>) | 'Not'(<Exp>)

π automaton semantics for π IR expressions I

- Recall that $\delta : L(G)^* \times L(G)^* \times Store \to Q$, and let $N, N_i \in \mathbb{N}$, $C, V \in L(G)^*$, $S \in Store$,

$$\delta(Num(N) :: C, V, S) = \delta(C, Num(N) :: V, S) \tag{11}$$

$$\delta(Sum(E_1, E_2) :: C, V, S) = \delta(E_1 :: E_2 :: \#SUM :: C, V, S) \tag{12}$$

$$\delta(\#SUM :: C, Num(N_1) :: Num(N_2) :: V, S) = \delta(C, (N_1 + N_2) :: V, S) \tag{13}$$

$$\ldots$$

$$\delta(Not(E) :: C, V, S) = \delta(E :: \#NOT :: C, V, S) \tag{14}$$

$$\delta(\#NOT :: C, Boo(true) :: V, S) = \delta(C, Boo(false) :: V, S) \tag{15}$$

$$\delta(\#NOT :: C, Boo(false) :: V, S) = \delta(C, Boo(true) :: V, S) \tag{16}$$

- Notation $h :: ls$ denotes the concatenation of element h with the list ls.
- C represents the *control* stack. V represents the *value* stack. S denotes the memory store.
- $\delta(\emptyset, V, S)$ denotes an *accepting state*.

π automaton semantics for π IR expressions II

- On a particular implementation of the π Framework, as in Python, <digits> denote built-in numbers in implementation language, such that all arithmetic operations, such as +, are defined. That's why N_i are in $\mathbb{N}$.

π IR commands

- Commands are language constructions that require a *memory* store to be evaluated.

  ⟨*Statement*⟩   ::= <Cmd>

  ⟨*Exp*⟩   := 'Id'(<String>)

  ⟨*Cmd*⟩   ::= 'Assign'(<Id>, <Exp>)
                   | 'Loop'(<BoolExp>, <Cmd>)
                   | 'CSeq'(<Cmd>, <Cmd>)

π automaton semantics for π IR commands I

- A location $l \in Loc$ denotes a memory cell.
- Storable and Bindable sets denote the data that may be mapped to by identifiers and locations on the memory and environment respectively.
- $Store = Id \mapsto Storable$, $Env = Loc \mapsto Bindable$, $Loc \subseteq Store$, $\mathbb{N} \subseteq Loc, Bindable$.
- Now the transition function is $\delta : L(G)^* \times L(G)^* \times Env \times Store \rightarrow Q$, and let $W \in String$, $C, V \in L(G)^*$, $S \in Store$, $E \in Env$, $B \in Bindable$, $l \in Loc$, $T \in Storable$, $X \in <Exp>$, $M, M_1, M_2 \in <Cmd>$, and

π automaton semantics for π IR commands II

expression $S' = S/[I \mapsto N]$ means that S' equals to S in all indices but I that is bound to N,

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S), \tag{17}$$

$$\text{where } E[W] = I \text{ and } S[I] = B,$$

$$\delta(Assign(W, X) :: C, V, E, S) = \delta(X :: \#ASSIGN :: C, W :: V, E, S'), \tag{18}$$

$$\delta(\#ASSIGN :: C, T :: W :: V, E, S) = \delta(C, V, E, S'), \tag{19}$$

$$\text{where } E[W] = I \text{ and } S' = S/[I \mapsto T],$$

$$\delta(Loop(X, M) :: C, V, E, S) = \delta(X :: \#LOOP :: C, Loop(X, M) :: V, E, S), \tag{20}$$

$$\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) = \delta(M :: Loop(X, M) :: C, V, E, S), \tag{21}$$

$$\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) = \delta(C, V, E, S), \tag{22}$$

$$\delta(CSeq(M_1, M_2) :: C, V, E, S) = \delta(M_1 :: M_2 :: C, V, E, S). \tag{23}$$

π IR declarations

- Declarations are statements that create an environment, binding identifiers to (bindable) values.
- In π IR, a bindable value is either a Boolean value, an integer or a location.
- From a syntactic standpoint, all classes are monotonically extended.

  $\langle Statement \rangle$ ::=   <Dec>

  $\langle Exp \rangle$ ::=   'Ref'(<Exp>)> | 'DeRef'(<Id>) | 'ValRef'(<Id>)

  $\langle Dec \rangle$ ::=   'Bind'(<Id>, <Exp>) | 'DSeq'(<Dec>, <Dec>)

  $\langle Cmd \rangle$ ::=   'Blk'(<Dec>, <Cmd>)

π automaton semantics for π IR declarations I

Let $BlockLocs = Set\{Loc\}$, now the transition function is
$\delta : L(G)^* \times L(G)^* \times Env \times Store \times BlockLocs \rightarrow Q$, and let $L, L' \in BlockLocs$,
$Loc \subseteq Storable$, and S/L means the store S without the locations in L,

π automaton semantics for π IR declarations II

$$\delta(Ref(X) :: C, V, E, S, L) = \delta(X :: \#REF :: C, V, E, S, L), \tag{24}$$

$$\delta(\#REF :: C, T :: V, E, S, L) = \delta(C, I :: V, E, S', L'), \text{ where } S' = S \cup [l \mapsto T], l \notin S, L' = L \cup \{l\}, \tag{25}$$

$$\delta(DeRef(Id(W)) :: C, V, E, S, L) = \delta(C, l :: V, E, S, L), \text{ where } l = E[W], \tag{26}$$

$$\delta(ValRef(Id(W)) :: C, V, E, S, L) = \delta(C, T :: V, E, S, L), \text{ where } T = S[S[E[W]]], \tag{27}$$

$$\delta(Bind(Id(W), X) :: C, V, E, S, L) = \delta(X :: \#BIND :: C, W :: V, E, S, L), \tag{28}$$

$$\delta(\#BIND :: C, B :: W :: E' :: V, E, S, L) = \delta(C, ([W \mapsto B] \cup E') :: V, E, S, L), \text{ where } E' \in \textbf{Env}, \tag{29}$$

$$\delta(\#BIND :: C, B :: W :: H :: V, E, S, L) = \delta(C, [W \mapsto B] :: H :: V, E, S, L), \text{ where } H \notin \textbf{Env}, \tag{30}$$

$$\delta(DSeq(D_1, D_2), X) :: C, V, E, S, L) = \delta(D_1 :: D_2 :: C, V, E, S, L), \tag{31}$$

$$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: M :: \#BLKCMD :: C, L :: V, E, S, \emptyset), \tag{32}$$

$$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(C, E :: V, E/E', S, L), \tag{33}$$

$$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L), \text{ where } S' = S/L. \tag{34}$$

π IR abstractions

- Abstractions extend Bindables by allowing a name to be bound to a list of formal parameters, a list of identifiers, and a block in the environment.
- Such names can be called and applied to actual parameters, a list of expressions.

  $\langle Dec \rangle$ ::= 'Bind'(<Id>, <Abs>)

  $\langle Abs \rangle$ ::= 'Abs'(<Formals>, <Blk>)

  $\langle Formals \rangle$ ::= <Id>*

  $\langle Cmd \rangle$ ::= 'Call'(<Id>, <Actuals>)

  $\langle Actuals \rangle$ ::= <Exp>*

π automaton semantics for π IR abstractions — Closures I

We chose a static binding semantics for abstractions. Therefore, we interpret abstractions as *closures* formed by an abstraction together with its declaration environment which defines the context in which the abstraction will be evaluated.

$$Closure : Formals \times Blk \times Env \rightarrow Bindable$$

π automaton semantics for π IR abstractions — Example I

```
 1  Imπ source code:
 2  # In this example we encapsulate the iterative calculation
 3  # of the factorial within a function call.
 4  let var z = 1
 5  in
 6      let fn f(x) =
 7          let var y = x
 8          in
 9              while not (y == 0)
10              do
11                  z := z * y
12                  y := y − 1
13      in f(10)
```

π automaton semantics for π IR abstractions — Example II

```
1  π IR AST:
2  Blk(Bind(Id(z), Ref(Num(1))),
3     Blk(BindAbs(Id(f), Abs(Id(x),
4        Blk(Bind(Id(y), Ref(Id(x))),
5           Loop(Not(Eq(Id(y), Num(0))),
6              CSeq(Assign(Id(z), Mul(Id(z), Id(y))),
7                   Assign(Id(y), Sub(Id(y), Num(1)))))))))),
8        Call(Id(f), Num(10))))
```

π automaton semantics for π IR abstractions I

Let $F \in Formals$, $B \in Blk$, $I \in Id$, $A \in Actuals$, $V_i \in Value$, $1 \leq i \leq n$, $n \in \mathbb{N}$,

$$\delta(\textbf{Abs}(F, B) :: C, V, E, S, L) = \delta(C, \textbf{Closure}(F, B, E) :: V, E, S, L) \tag{35}$$

$$\delta(\textbf{Call}(I, [X_1, X_2, \ldots, X_n])) :: C, V, E, S, L) = \tag{36}$$
$$\delta(X_n :: X_{n-1} :: \ldots :: X_1 :: \#\textbf{CALL}(I, n) :: C, V, E, S, L)$$
$$\delta(\#\textbf{CALL}(I, n) :: C, [V_1, V_2, \ldots, V_n] :: V, [I \mapsto \textbf{Closure}(F, B, E_1)]E_2, S, L) = \tag{37}$$
$$\delta(B :: \#BLKCMD :: C, E_2 :: V, (E_1 / \textbf{match}(F, [V_1, V_2, \ldots, V_n])), S, L)$$

$$\textbf{match} : \textbf{Id}^* \times \textbf{Values}^* \to \textbf{Env}$$
$$\textbf{match}(fl, al) = \textbf{if } |fl| \neq |al| \textbf{ than } \{\} \textbf{ else } \_\textbf{match}(fl, al, \{\})$$
$$\_\textbf{match} : \textbf{Id}^* \times \textbf{Values}^* \times \textbf{Env} \to \textbf{Env}$$
$$\_\textbf{match}([], [], E) = E$$
$$\_\textbf{match}(f, a, E) = \{f \mapsto a\}E$$
$$\_\textbf{match}(f :: fl, a :: al, E) = \_\textbf{match}(fl), al, \{f \mapsto a\}E)$$

π IR recursive abstractions

- Abstractions can be recursive to allow for the declaration of recursive functions.

  $\langle Dec \rangle$ ::= 'Rbnd'(<Id>, <Abs>)

π automaton semantics for π IR recursive abstractions — Recursive closures I

In the context of static binding semantics for abstractions, in a call to a recursive function, the evaluation of identifiers needs to be reminded about the binding of the function name to a closure.

$$Rec : Formals \times Blk \times Env \times Env \rightarrow Bindable$$
$$unfold : Env \rightarrow Env$$
$$reclose_E : Env \rightarrow Env$$

π automaton semantics for π IR recursive abstractions — Recursive closures II

$$unfold(E) = reclose_E(E) \tag{38}$$

$$reclose_E(I \mapsto Closure(F, B, E')) = (I \mapsto Rec(F, B, E', E)) \tag{39}$$

$$reclose_E(I \mapsto Rec(F, B, E', E'')) = (I \mapsto Rec(F, B, E', E)) \tag{40}$$

$$reclose_E(I \mapsto v) = (I \mapsto v) \text{ if } v \neq Closure(F, B, E) \tag{41}$$

$$reclose_E(E_1 \cup E_2) = reclose_E(E_1) \cup reclose_E(E_2) \tag{42}$$

$$reclose_E(\emptyset) = \emptyset \tag{43}$$

π automaton semantics for π IR recursive abstractions —
Recursive closures I

$$\delta(\boldsymbol{Rbnd}(I, Abs(F, B)) :: C, V, E, S, L) = \delta(C, \boldsymbol{unfold}(I \mapsto \boldsymbol{Closure}(F, B, E)) :: V, E, S, L) \tag{44}$$

$$\delta(\#\boldsymbol{CALL}(I, n) :: C, [V_1, V_2, ..., V_n] :: V, [I \mapsto \boldsymbol{Rec}(F, B, E_1, E_2)] \cup E_3, S, L) = \tag{45}$$
$$\delta(B :: \#\boldsymbol{BLKCMD} :: C, E_2 :: V, E_1 / \boldsymbol{match}(F, [V_1, V_2, ..., V_n]), S, L)$$

π IR expressions in Python I

```
1  class Statement:
2      def __init__(self, *args):
3          self.opr =args
4      def __str__(self):
5          ret =str(self.__class__.__name__)+"("
6          for o in self.opr:
7              ret +=str(o)
8          ret +=")"
9          return ret
10 class Exp(Statement): pass
11 class ArithExp(Exp): pass
```

π IR expressions in Python II

```python
1  class Num(ArithExp):
2      def __init__(self, f):
3          assert(isinstance(f, int))
4          ArithExp.__init__(self,f)
5  class Sum(ArithExp):
6      def __init__(self, e1, e2):
7          assert(isinstance(e1, Exp) and isinstance(e2, Exp))
8          ArithExp.__init__(self, e1, e2)
9  ...
```

π IR expressions in Python III

```
1 class BoolExp(Exp): pass
2 class Eq(BoolExp):
3     def __init__(self, e1, e2):
4         assert(isinstance(e1, Exp) and isinstance(e2, Exp))
5         BoolExp.__init__(self, e1, e2)
6 ...
```

π IR expressions in Python IV

```
1  exp =Sum(Num(1), Mul(Num(2), Num(4)))
2  print(exp)
3
4  Sum(Num(1)Mul(Num(2)Num(4)))
```

π IR expressions in Python V

```
1 exp2 =Mul(2, 1)
2 ---------------------------------------------------------------------
                                  ---
3 AssertionError Traceback (most recent call last)
4 <ipython-input-7-00fd40a79a54> in <module>()
5 ---->1 exp2 =Mul(2, 1)
6
7 <ipython-input-5-42a82e58862f> in __init__(self, e1, e2)
8      28 class Mul(ArithExp):
9      29 def __init__(self, e1, e2):
10 --->30 assert(isinstance(e1, Exp) and isinstance(e2, Exp))
11      31 ArithExp.__init__(self, e1, e2)
12      32 class BoolExp(Exp): pass
13
14 AssertionError:
```

π automaton for π IR expressions I

```python
## Expressions
class ValueStack(list): pass
class ControlStack(list): pass
class ExpKW:
    SUM ="#SUM"
    SUB ="#SUB"
    MUL ="#MUL"
    EQ ="#EQ"
    NOT ="#NOT"
```

π automaton for π IR expressions II

```
1  class ExpPiAut(dict):
2      def __init__(self):
3          self["val"] =ValueStack()
4          self["cnt"] =ControlStack()
5      def __evalSum(self, e):
6          e1 =e.opr[0]
7          e2 =e.opr[1]
8          self.pushCnt(ExpKW.SUM)
9          self.pushCnt(e1)
10         self.pushCnt(e2)
11     def pushCnt(self, e):
12         cnt =self.cnt()
13         cnt.append(e)
14 ...
```

π automaton for π IR expressions III

```
1  ea =ExpPiAut()
2  print(exp)
3  ea.pushCnt(exp)
4  while not ea.emptyCnt():
5      ea.eval()
6      print(ea)
```

π automaton for π IR expressions IV

```
1  Sum(Num(1)Mul(Num(2)Num(4)))
2  {'val': [], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, <
                                 __main__.Mul object at 0x1118516d8>]
                                 }
3  {'val': [], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#MUL'
                                 , <__main__.Num object at
                                 0x111851630>, <__main__.Num object
                                 at 0x1118516a0>]}
4  {'val': [4], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#MUL
                                 ', <__main__.Num object at
                                 0x111851630>]}
5  {'val': [4, 2], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>, '#
                                 MUL']}
6  {'val': [8], 'cnt': ['#SUM', <__main__.Num object at 0x111851470>]}
7  {'val': [8, 1], 'cnt': ['#SUM']}
8  {'val': [9], 'cnt': []}
```

π IR commands I

```python
1  class Cmd(Statement): pass
2  class Id(Exp):
3      def __init__(self, s):
4          assert(isinstance(s, str))
5          Exp.__init__(self, s)
6  class Assign(Cmd):
7      def __init__(self, i, e):
8          assert(isinstance(i, Id) and isinstance(e, Exp))
9          Cmd.__init__(self, i, e)
10 class Loop(Cmd):
11     def __init__(self, be, c):
12         assert(isinstance(be, BoolExp) and isinstance(c, Cmd))
13         Cmd.__init__(self, be, c)
14 class CSeq(Cmd):
15     def __init__(self, c1, c2):
16         assert(isinstance(c1, Cmd) and isinstance(c2, Cmd))
17         Cmd.__init__(self, c1, c2)
```

π IR commands II

```
1  cmd =Assign(Id("x"), Num(1))
2  print(type(cmd))
3  print(cmd)
4  <class '__main__.Assign'>
5  Assign(Id(x)Num(1))
```

π automaton for π IR commands I

Environment, Location, Store and commands opcodes.

```
1  ## Commands
2  class Env(dict): pass
3  class Loc(int): pass
4  class Sto(dict): pass
5  class CmdKW:
6      ASSIGN ="#ASSIGN"
7      LOOP ="#LOOP"
```

π automaton for π IR commands II

π automaton for commands extends the π automaton for expressions.

```python
class CmdPiAut(ExpPiAut):
    def __init__(self):
        self["env"] =Env()
        self["sto"] =Sto()
        ExpPiAut.__init__(self)
    def env(self):
        return self["env"]
    def getLoc(self, i):
        en =self.env()
        return en[i]
    def sto(self):
        return self["sto"]
    def updateStore(self, l, v):
        st =self.sto()
        st[l] =v
```

π automaton for π IR commands III
π semantics for assignment.

$$\delta(Assign(W,X) :: C, V, E, S) = \delta(X :: \#ASSIGN :: C, W :: V, E, S'),$$
$$\delta(\#ASSIGN :: C, T :: W :: V, E, S) = \delta(C, V, E, S'),$$
$$\text{where } E[W] = l \text{ and } S' = S/[l \mapsto T].$$

```
1    def __evalAssign(self, c):
2        i =c.opr[0]
3        e =c.opr[1]
4        self.pushVal(i.opr[0])
5        self.pushCnt(CmdKW.ASSIGN)
6        self.pushCnt(e)
7    def __evalAssignKW(self):
8        v =self.popVal()
9        i =self.popVal()
10       l =self.getLoc(i)
11       self.updateStore(l, v)
```

π automaton for π IR commands IV

π semantics for identifiers.

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S),$$
$$\text{where } E[W] = l \text{ and } S[l] = B.$$

```
1    def __evalId(self, i):
2        s =self.sto()
3        l =self.getLoc(i)
4        self.pushVal(s[l])
```

π automaton for π IR commands V

π semantics for loop: recursive step.

$$\delta(Loop(X,M) :: C, V, E, S) = \delta(X :: \#LOOP :: C, Loop(X,M) :: V, E, S).$$

```
1    def __evalLoop(self, c):
2        be =c.opr[0]
3        bl =c.opr[1]
4        self.pushVal(Loop(be, bl))
5        self.pushVal(bl)
6        self.pushCnt(CmdKW.LOOP)
7        self.pushCnt(be)
```

π automaton for π IR commands VI

π semantics for loop: basic steps.

$$\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) = \delta(M :: Loop(X, M) :: C, V, E, S),$$
$$\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) = \delta(C, V, E, S).$$

```
1    def __evalLoopKW(self):
2        t =self.popVal()
3        if t:
4            c =self.popVal()
5            lo =self.popVal()
6            self.pushCnt(lo)
7            self.pushCnt(c)
8        else:
9            self.popVal()
10           self.popVal()
```

Computação

π automaton for π IR commands VII

π semantics for command composition.

$$\delta(CSeq(M_1, M_2) :: C, V, E, S) = \delta(M_1 :: M_2 :: C, V, E, S).$$

```
1    def __evalCSeq(self, c):
2        c1 =c.opr[0]
3        c2 =c.opr[1]
4        self.pushCnt(c2)
5        self.pushCnt(c1)
```

π automaton for π IR commands VIII

Commands are now on the top of the food chain.

```
1    def eval(self):
2        c =self.popCnt()
3        if isinstance(c, Assign):
4            self.__evalAssign(c)
5        elif c ==CmdKW.ASSIGN:
6            self.__evalAssignKW()
7        elif isinstance(c, Id):
8            self.__evalId(c.opr[0])
9        elif isinstance(c, Loop):
10           self.__evalLoop(c)
11       elif c ==CmdKW.LOOP:
12           self.__evalLoopKW()
13       elif isinstance(c, CSeq):
14           self.__evalCSeq(c)
15       else:
16           self.pushCnt(c)
17           ExpPiAut.eval(self)
```

π IR declarations in Python I

DeRef and ValRef not implemented yet.

```
1  ## Declarations
2  class Dec(Statement): pass
3  class Bind(Dec):
4     def __init__(self, i, e):
5         assert (isinstance(i, Id) and isinstance(e, Exp))
6         Dec.__init__(self, i, e)
7  class Ref(Exp):
8     def __init__(self, e):
9         assert (isinstance(e, Exp))
10        Exp.__init__(self, e)
11 class Cns(Exp):
12    def __init__(self, e):
13        assert (isinstance(e, Exp))
14        Exp.__init__(self, e)
15 class Blk(Cmd):
16    def __init__(self, d, c):
```

π IR declarations in Python II

```
17        assert (isinstance(d, Dec) and isinstance(c, Cmd))
18        Cmd.__init__(self, d, c)
```

π IR declarations in Python III

```python
1  class DSeq(Dec):
2      def __init__(self, d1, d2):
3          assert (isinstance(d1, Dec) and isinstance(d2, Dec))
4          Dec.__init__(self, d1, d2)
```

π IR declarations in Python IV

```
1  ## Declarations
2  class DecExpKW(ExpKW):
3     REF ="#REF"
4     CNS ="#CNS"
5
6  class DecCmdKW(CmdKW):
7     BLKDEC ="#BLKDEC"
8     BLKCMD ="#BLKCMD"
9
10 class DecKW:
11    BIND ="#BIND"
12    DSEQ ="#DSEQ"
13
14 class DecPiAut(CmdPiAut):
15    def __init__(self):
16       self["locs"] =[]
17       CmdPiAut.__init__(self)
18
```

π IR declarations in Python V

```python
19     def locs(self):
20         return self["locs"]
21
22     def pushLoc(self, l):
23         ls =self.locs()
24         ls.append(l)
25
26     def __evalRef(self, e):
27         ex =e.opr[0]
28         self.pushCnt(DecExpKW.REF)
29         self.pushCnt(ex)
30
31     def __newLoc(self):
32         sto =self.sto()
33         if sto:
34             return max(list(sto.keys())) +1
35         else:
36             return 0.0
37
```

π IR declarations in Python VI

```
38    def __evalRefKW(self):
39        v =self.popVal()
40        l =self.__newLoc()
41        self.updateStore(l, v)
42        self.pushLoc(l)
43        self.pushVal(l)
44
45    def __evalBind(self, d):
46        i =d.opr[0]
47        e =d.opr[1]
48        self.pushVal(i)
49        self.pushCnt(DecKW.BIND)
50        self.pushCnt(e)
51
52    def __evalBindKW(self):
53        l =self.popVal()
54        i =self.popVal()
55        x =i.opr[0]
56        self.pushVal({x: l})
```

π IR declarations in Python VII

```
57
58    def __evalDSeq(self, ds):
59        d1 =ds.opr[0]
60        d2 =ds.opr[1]
61        self.pushCnt(DecKW.DSEQ)
62        self.pushCnt(d2)
63        self.pushCnt(d1)
64
65    def __evalDSeqKW(self):
66        d2 =self.popVal()
67        d1 =self.popVal()
68        d1.update(d2)
69        self.pushVal(d1)
70
71    def __evalBlk(self, d):
72        ld =d.opr[0]
73        c =d.opr[1]
74        l =self.locs()
75        self.pushVal(list(l))
```

π IR declarations in Python VIII

```
76        self.pushVal(c)
77        self.pushCnt(DecCmdKW.BLKDEC)
78        self.pushCnt(ld)
79
80    def __evalBlkDecKW(self):
81        d =self.popVal()
82        c =self.popVal()
83        l =self.locs()
84        self.pushVal(l)
85        en =self.env()
86        ne =en.copy()
87        ne.update(d)
88        self.pushVal(en)
89        self["env"] =ne
90        self.pushCnt(DecCmdKW.BLKCMD)
91        self.pushCnt(c)
92
93    def __evalBlkCmdKW(self):
94        en =self.popVal()
```

π IR declarations in Python IX

```
95      ls =self.popVal()
96      self["env"] =en
97      s =self.sto()
98      s ={k: v for k, v in s.items() if k not in ls}
99      self["sto"] =s
00      # del ls
01      ols =self.popVal()
02      self["locs"] =ols
03
04  def eval(self):
05      d =self.popCnt()
06      if isinstance(d, Bind):
07          self.__evalBind(d)
08      elif d ==DecKW.BIND:
09          self.__evalBindKW()
10      elif isinstance(d, DSeq):
11          self.__evalDSeq(d)
12      elif d ==DecKW.DSEQ:
13          self.__evalDSeqKW()
```

π IR declarations in Python X

```
14        elif isinstance(d, Ref):
15            self.__evalRef(d)
16        elif d ==DecExpKW.REF:
17            self.__evalRefKW()
18        elif isinstance(d, Blk):
19            self.__evalBlk(d)
20        elif d ==DecCmdKW.BLKDEC:
21            self.__evalBlkDecKW()
22        elif d ==DecCmdKW.BLKCMD:
23            self.__evalBlkCmdKW()
24        else:
25            self.pushCnt(d)
26            CmdPiAut.eval(self)
```

π IR declarations in Python XI

```
1  dc =DecPiAut()
2  fac =Loop(Not(Eq(Id("y"), Num(0))),
3           CSeq(Assign(Id("x"), Mul(Id("x"), Id("y"))),
4                Assign(Id("y"), Sub(Id("y"), Num(1)))))
5  dec =DSeq(Bind(Id("x"), Ref(Num(1))),
6           Bind(Id("y"), Ref(Num(200))))
7  fac_blk =Blk(dec, fac)
8  dc.pushCnt(fac_blk)
9  while not dc.emptyCnt():
10     aux =dc.copy()
11     dc.eval()
12     if dc.emptyCnt():
13         print(aux)
```

π IR abstractions in Python I

```python
class Formals(list):
    def __init__(self, f):
        if isinstance(f, list):
            for a in f:
                if not isinstance(a, Id):
                    raise IllFormed(self, a)
            self.append(f)
        else:
            raise IllFormed(self, f)

class Abs:
    def __init__(self, f, b):
        if isinstance(f, list):
            if isinstance(b, Blk):
                self._opr =[f, b]
            else:
                raise IllFormed(self, b)
```

π IR abstractions in Python II

```
19          else:
20              raise IllFormed(self, f)
21
22      def formals(self):
23          return self._opr[0]
24
25      def blk(self):
26          return self._opr[1]
27
28      def __str__(self):
29          ret =str(self.__class__.__name__) +"("
30          formals =self.formals()
31          ret +=str(formals[0]) # First formal argument
32          for i in range(1, len(formals)):
33              ret +=", "
34              ret +=str(formals[i]) # Remaining formal arguments
35          ret +=", "
36          ret +=str(self.blk()) # Abstraction block
37          ret +=")"
```

π IR abstractions in Python III

```
38          return ret
39
40  class BindAbs(Bind):
41      '''
42      BindAbs is a form of bind but that receives an Abs instead of an
43      expression.
44      '''
45      def __init__(self, i, p):
46          if isinstance(i, Id):
47              if isinstance(p, Abs):
48                  Dec.__init__(self, i, p)
49              else:
50                  raise IllFormed(self, p)
51          else:
52              raise IllFormed(self, i)
53
54  class Actuals(list):
55      def __init__(self, a):
56          if isinstance(a, list):
```

π IR abstractions in Python IV

```
57              for e in a:
58                  if not isinstance(e, Exp):
59                      raise IllFormed(self, e)
60              self.append(a)
61          else:
62              raise IllFormed(self, a)
63
64  class Call(Cmd):
65      def __init__(self, f, actuals):
66          if isinstance(f, Id):
67              if isinstance(actuals, list):
68                  Cmd.__init__(self, f, actuals)
69              else:
70                  raise IllFormed(self, actuals)
71          else:
72              raise IllFormed(self, f)
73
74      def caller(self):
75          return self.operand(0)
```

π IR abstractions in Python V

```
76
77     def actuals(self):
78         return self.operand(1)
79
80 class Closure(dict):
81     def __init__(self, f, b, e):
82         if isinstance(f, list):
83             if isinstance(b, Blk):
84                 # I wanted to write assert(isinstance(e, Env)) but it
                                                        fails.
85                 if isinstance(e, dict):
86                     self["for"] =f # Formal parameters
87                     self["env"] =e # Current environment
88                     self["block"] =b # Procedure block
89                 else:
90                     raise IllFormed(self, e)
91             else:
92                 raise IllFormed(self, b)
93         else:
```

π IR abstractions in Python VI

```
94            raise IllFormed(self, f)
95
96    def __str__(self):
97        ret =str(self.__class__.__name__) +"("
98        formals =self.formals()
99        fst_formal =formals[0] # First formal argument
00        ret +=str(fst_formal)
01        for i in range(1, len(formals)):
02            ret +=", "
03            formal =formals[i] # Remaining formal arguments
04            ret +=str(formal)
05        ret +=", "
06        ret +=str(self.blk()) # Closure block
07        ret +=")"
08        return ret
09
10    def formals(self):
11        return self['for']
12
```

π IR abstractions in Python VII

```
13      def env(self):
14          return self['env']
15
16      def blk(self):
17          return self['block']
18
19  class AbsPiAut(DecPiAut):
20      def __evalAbs(self, a):
21          if not isinstance(a, Abs): # p must be an abstraction
22              raise EvaluationError(self, "Function __evalAbs called with
                                           no abstraction but with "
                                           , a, " instead.")
23          else:
24              f =a.formals() # Formal parameters
25              b =a.blk() # Body
26              e =self.env() # Current environment
27              # Closes the given abs. with the current env
28              c =Closure(f, b, e)
29              # Closure c is pushed to the value stack such that
```

π IR abstractions in Python VIII

```
30            self.pushVal(c)
31            # a BIND may create a new binding to a given identifier.
32
33    def __match(self, f, a):
34        '''
35        Given a list of formal parameters and a list of actual parameters,
36        it returns an environment relating the elements of the former
                                              with the latter.
37        '''
38        if isinstance(f, list):
39            if isinstance(a, list):
40                if len(f) ==0:
41                    return {}
42                if len(f) ==len(a) and len(f) >0:
43                # For some reason, f[0] is a tuple, not an Id.
44                    f0 =f[0]
45                    a0 =a[0]
46                    b0 ={f0.id(): a0.num()}
```

π IR abstractions in Python IX

```
47              if len(f) ==1:
48                  return b0
49              else:
50                  # For some reason, f[0] is a tuple, not an Id.
51                  f1 =f[1]
52                  a1 =a[1]
53                  b1 ={f1.id(): a1.num()}
54                  e =b0.update(b1)
55                  for i in range(2, len(f)):
56                      fi =f[i][0]
57                      ai =a[i][0]
58                      e.update({fi.id(): ai.num()})
59                  return e
60          else:
61              raise EvaluationError("Call to '__match' on " +str(self) +
                                          ": " +"formals and
                                          actuals differ in
                                          size.")
62      else:
```

π IR abstractions in Python X

```
63              raise EvaluationError("Call to '__match' on " +str(self) +":
                                            " +" no formals, but with
                                              ", f, " instead.")
64
65    def __evalCall(self, c):
66        '''
67        Essentially, a call is translated into a block.
68        If we were progrmming pi in a symbolic language,
69        we could simply crete a proper block and push it to the control
                                            stack.
70        However, the environment is not symbolic: is a dictionary of
                                            objects.
71        To create a block we would need to "pi-IR-fy" it, that is,
                                            recreate the
72        pi IR tree from the concrete environmnet and joint it with
                                            matches created
73        also at pi IR level. These would be pushed back into the control
                                            stack and
```

π IR abstractions in Python XI

```
74        reobjectifyed. Thus, to avoid pi-IRfication and reevaluatuation
                                      of the
75        environment we manipulate it at the object level, which is
                                      dangerous but
76        seems to be correct.
77
78        In this implementation, actual parameters are already evaluated.
79        '''
80        if not isinstance(c, Call): # c must be a Call object
81            raise EvaluationError("Call to __evalCall with no Call object
                                          but with ", c, " instead
                                          .")
82        else:
83            # Procedure to be called
84            caller =c.caller()
85            # Retrieves the current environment.
86            e =self.env()
87            # Retrieves the closure associated with the caller function.
88            clos =e[caller.id()]
```

π IR abstractions in Python XII

```
89          # Retrieves the actual parameters from the call.
90          a =c.actuals()
91          # Retrieves the formal parameters from the closure.
92          f =clos.formals()
93          # Matches formals and actuals, creating an environment.
94          d =self.__match(f, a)
95          # Retrives the closure's environment.
96          ce =clos.env()
97          # The caller's block must run on the closures environment
98          # overwritten with the matches.
99          d.update(ce)
00          self["env"] =d
01          self.pushVal(self.locs())
02          # Saves the current environment in the value stack.
03          self.pushVal(e)
04          # Pushes the keyword BLKCMD for block completion.
05          self.pushCnt(DecCmdKW.BLKCMD)
06          # Pushes the body of the caller function into the control
                                                    stack.
```

π IR abstractions in Python XIII

```
07              self.pushCnt(clos.blk())
08
09    def eval(self):
10        d =self.popCnt()
11        if isinstance(d, Abs):
12            self.__evalAbs(d)
13        elif isinstance(d, Call):
14            self.__evalCall(d)
15        else:
16            self.pushCnt(d)
17            DecPiAut.eval(self)
```

Complete Imp grammar in EBNF notation I

```
 1  @@grammar::IMP
 2  @@eol_comments ::/#.*?/ start = @ : cmd ;
 3
 4
 5
 6  cmd =nop | let | assign | loop | call ;
 7
 8  call =i:identifier '(' { a:actual }* ')' ;
 9
10  actual =e1:expression { ',' e2:expression }* | {} ;
11
12  nop ='nop' ;
13
14  loop =op:'while' ~ e:expression 'do' { c:cmd }+ ;
15
16  assign =id:identifier op:':=' ~ e:expression ;
17
18  let =op:'let' ~ d:dec 'in' { c:cmd }+ ;
```

Complete Imp grammar in EBNF notation II

```
19
20 dec =var | fn ;
21
22 var =op:'var' ~ id:identifier '=' e:expression ;
23
24 fn =op:'fn' ~ id:identifier '(' f:formal ')' '=' c:cmd ;
25
26 formal =i1:identifier { ',' i2:identifier }* | {} ;
27
28 expression =@:bool_expression ;
29
30 bool_expression =negation | equality | conjunction | disjunction
31                  | lowereq | greatereq | lowerthan | greaterthan
32                  | add_expression ;
33
34 equality =left:add_expression op:"==" ~ right:bool_expression ;
35
36 conjunction =left:add_expression op:"and" ~ right:bool_expression ;
37
```

Complete Imp grammar in EBNF notation III

```
38  disjunction =left:add_expression op:"or" ~ right:bool_expression ;
39
40  lowereq =left:add_expression op:"<=" ~ right:add_expression ;
41
42  greatereq =left:add_expression op:">=" ~ right:add_expression ;
43
44  lowerthan =left:add_expression op:"<" ~ right:add_expression ;
45
46  greaterthan =left:add_expression op:">" ~ right:add_expression ;
47
48  parentesisexp ='(' ~ @:bool_expression ')' ;
49
50  negation =op:'not' ~ b:bool_expression ;
51
52  add_expression =addition | subtraction | @:mult_expression ;
53
54  addition =left:mult_expression op:"+" ~ right:add_expression ;
55
56  subtraction =left:mult_expression op:"-" ~ right:add_expression ;
```

Complete Imp grammar in EBNF notation IV

```
57
58 mult_expression =multiplication | division
59                 | atom
60                 | parentesisexp ;
61
62 multiplication =left:atom op:"*" ~ right:mult_expression ;
63
64 division =left:atom op:"/" ~ right:mult_expression ;
65
66 atom =number | truth | identifier ;
67
68 number =/\d+/ ;
69
70 identifier =/(?!\d)\w+/ ;
71
72 truth ='True' | 'False' ;
```

Example: iterative factorial

```
1  # The classic iterative factorial example
2  let var z = 1
3  in
4      let var y = 10
5      in
6          while not (y == 0)
7          do
8              z := z * y
9              y := y − 1
```

Example: iterative factorial within a function

```
1  # In this example we encapsulate the iterative calculation
2  # of the factorial within a function call.
3  let var z = 1
4  in
5      let fn f(x) =
6          let var y = x
7          in
8              while not (y == 0)
9              do
10                  z := z * y
11                  y := y − 1
12      in f(10)
```