Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

November 30, 2018

http://github.com/ChristianoBraga/PiFramework

Compiler pipeline

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

Compiler pipeline and formal languages

$$\begin{array}{c}source\\code\end{array} \xrightarrow{\substack{Regular\\Grammar\\lexer}} tokens \xrightarrow{\substack{ContextFree\\Grammar\\parser}} \begin{array}{c}concrete\\syntax\\tree\end{array} \xrightarrow{\substack{ContextFree\\Grammar\\AST\ \ transformer}} \begin{array}{c}abstract\\syntax\\tree\end{array} \xrightarrow{\substack{ContextSensitive\\Grammar\\type\ \ checker}} \begin{array}{c}abstract\\syntax\\tree\end{array} \xrightarrow{\substack{Turing\\Machine\\code\ \ generator}} \begin{array}{c}machine\\code\end{array} \xrightarrow{\substack{Turing\\Machine\\optimizer}} \begin{array}{c}optimized\\machine\\code\end{array}$$

Compiler pipeline with the π Framework

$$\mathit{source\ code} \xrightarrow{\underbrace{\mathit{lexer} \circ \mathit{parser} \circ \mathit{AST\ transformer} \circ \mathit{type\ checker} \circ \mathit{code\ generator}}_{}} \pi\ \mathit{lib}$$

Chomsky's hierarchy

$\mathit{source\ code} \xrightarrow{\mathit{lexer}} \circ \xrightarrow{\mathit{parser}} \circ \xrightarrow[\mathit{transformer}]{\mathit{AST}} \circ \xrightarrow[\mathit{checker}]{\mathit{type}} \circ \xrightarrow[\mathit{generator}]{\mathit{code}} \pi\ \mathit{lib}$

π Automata

$\pi\ \mathit{lib} \xrightarrow{\mathit{interpreter}} \mathit{output}$

$\pi\ \mathit{lib} \xrightarrow{\mathit{code\ generator}} \mathit{machine\ code}$

$\pi\ \mathit{lib} \xrightarrow{\mathit{model\ checker}(P)} \mathit{counter\text{-}examples}$

where P is a property in a suitable logic.

- π lib defines a set of constructions common to many programming languages.
- π lib 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 lib 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

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

$$\langle \mathit{digits} \rangle ::= \langle \mathit{digit} \rangle^{+}$$

$$\langle \mathit{boolean} \rangle ::= \text{`}\texttt{true}\text{'} \mid \text{`}\texttt{false}\text{'}$$

A calculator: concrete syntax

$$
\begin{array}{lcl}
\langle exp \rangle & ::= & \langle aexp \rangle \mid \langle bexp \rangle \\[1em]
\langle aexp \rangle & ::= & \langle aexp \rangle \text{ `+' } \langle term \rangle \mid \langle aexp \rangle \text{ `-' } \langle term \rangle \mid \langle term \rangle \\[1em]
\langle term \rangle & ::= & \langle term \rangle \text{ `}*\text{' } \langle factor \rangle \mid \langle term \rangle \text{ `/' } \langle factor \rangle \mid \langle factor \rangle \\[1em]
\langle factor \rangle & ::= & \text{`(' } \langle aexp \rangle \text{ `)' } \mid \langle digits \rangle \\[1em]
\langle bexp \rangle & ::= & \langle boolean \rangle \mid \text{ `\~{}' } \langle bexp \rangle \mid \langle bexp \rangle \langle boolop \rangle \langle bexp \rangle \\
 & \mid & \langle aexp \rangle \langle iop \rangle \langle aexp \rangle \\[1em]
\langle boolop \rangle & ::= & \text{`=' } \mid \text{ `/\textbackslash' } \mid \text{ `\textbackslash/'} \\[1em]
\langle iop \rangle & ::= & \text{`<' } \mid \text{ `>' } \mid \text{ `<=' } \mid \text{ `>='}
\end{array}
$$

Instituto de Computação

A calculator: abstract syntax

$$\langle exp \rangle ::= \langle digits \rangle \mid \langle boolean \rangle \mid \langle exp \rangle\ \langle bop \rangle\ \langle exp \rangle$$

$$\langle bop \rangle ::= \text{`+'} \mid \text{`-'} \mid \text{`}*\text{'} \mid \text{`/'} \mid \text{`='} \mid \text{`\textbackslash/'} \mid \text{`/\textbackslash'} \mid \text{`<'} \mid \text{`>'} \mid \text{`<='} \mid \text{`>='}$$

A calculator: π denotations I

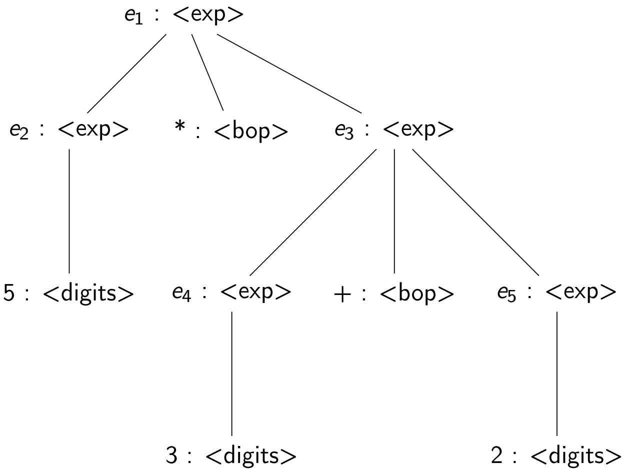
Let D in <digits>, B in <boolean> and E_1, E_2 in <exp>,

$$[\![D]\!]_\pi = Num(D) \tag{1}$$
$$[\![B]\!]_\pi = Boo(B) \tag{2}$$
$$[\![E_1 + E_2]\!]_\pi = Sum([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{3}$$
$$[\![E_1 - E_2]\!]_\pi = Sub([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{4}$$
$$[\![E_1 * E_2]\!]_\pi = Mul([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{5}$$
$$[\![E_1 / E_2]\!]_\pi = Div([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{6}$$
$$[\![E_1 < E_2]\!]_\pi = Lt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{7}$$
$$[\![E_1 <= E_2]\!]_\pi = Le([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{8}$$
$$[\![E_1 > E_2]\!]_\pi = Gt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{9}$$
$$[\![E_1 >= E_2]\!]_\pi = Ge([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{10}$$

A calculator: π denotations II

- π denotations are *functions* $[\![\cdot]\!]_\pi : AST \to \pi\ lib$, where AST denotes the *datatype* for the abstract syntax tree and $\pi\ lib$ denotes the datatype for π *lib* programs.
- Note that $[\![\cdot]\!]_\pi$ has *trees* as parameters, instances of AST. The example expression $5 * (3 + 2)$ becomes

A calculator: π denotations III



A calculator: π denotations IV

$$
\begin{aligned}
[\![5 * (3+2)]\!]_\pi &= Mul([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) && \textit{by Equation 5}\\
Mul([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) &= Mul(Num(5), [\![(3+2)]\!]_\pi) && \textit{by Equations 1}\\
Mul(num(5), [\![(3+2)]\!]_\pi) &= Mul(Num(5), Sum([\![3]\!]_\pi, [\![2]\!]_\pi) && \textit{by Equation 3}\\
Mul(Num(5), Sum([\![3]\!]_\pi, [\![2]\!]_\pi) &= Mul(Num(5), Sum(Num(3), [\![2]\!]_\pi) && \textit{by Equation 1}\\
Mul(Num(5), Sum(Num(3), [\![2]\!]_\pi) &= Mul(Num(5), Sum(Num(3), Num(2)) && \textit{by Equation 1}
\end{aligned}
$$

A calculator: executing π lib 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 $[s_1, s_2, \ldots, s_n]$.)

$$
\begin{aligned}
\delta([Mul(Num(5), Sum(Num(3), Num(2)], \emptyset, \emptyset) &= \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset)\\
\delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset) &= \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset)\\
\delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset) &= \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset)\\
\delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset) &= \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset)\\
\delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset) &= \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset)\\
\delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset) &= \delta([\#MUL], [Num(5), Num(5)], \emptyset)\\
\delta([\#MUL], [Num(5), Num(5)], \emptyset) &= \delta(\emptyset, [Num(25)], \emptyset)\\
\delta(\emptyset, [Num(25)], \emptyset) &= Num(25)
\end{aligned}
$$

Excerpt of π lib expressions

⟨*Statement*⟩ ::= <Exp>

⟨*Exp*⟩ ::= <ArithExp> | <BoolExp>

⟨*ArithExp*⟩ ::= ‘Num’(<digits>) | ‘Sum’(<Exp> , <Exp>) |
‘Sub’(<Exp>, <Exp>) | ‘Mul’(<Exp>, <Exp>)

⟨*BoolExp*⟩ ::= ‘Eq’(<Exp>, <Exp>) | ‘Not’(<Exp>)

π automaton semantics for π lib expressions

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

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

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

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

$$\ldots$$

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

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

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

- 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 lib commands

- Commands are language constructions that require both an *environment* and a *memory* store to be evaluated.

  ⟨*Statement*⟩ ::= <Cmd>

  ⟨*Exp*⟩ ::= 'Id'(<String>)

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

- From a syntactic standpoint, they extend both statements and expressions, as an identifier is an expression.

π automaton semantics for π lib 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 $\delta : L(G)^* \times L(G)^* \times Env \times Store \to Q$, 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 lib commands II

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

$$
\begin{aligned}
\delta(Id(W) :: C, V, E, S) &= \delta(C, B :: V, E, S), && (17)\\
&\quad \textbf{where } E[W] = I \textbf{ and } S[I] = B, \\
\delta(Assign(W, X) :: C, V, E, S) &= \delta(X :: \#ASSIGN :: C, W :: V, E, S'), && (18)\\
\delta(\#ASSIGN :: C, T :: W :: V, E, S) &= \delta(C, V, E, S'), && (19)\\
&\quad \textbf{where } E[W] = I \textbf{ and } S' = S/[I \mapsto T], \\
\delta(Loop(X, M) :: C, V, E, S) &= \delta(X :: \#LOOP :: C, Loop(X, M) :: V, E, S), && (20)\\
\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) &= \delta(M :: Loop(X, M) :: C, V, E, S), && (21)\\
\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) &= \delta(C, V, E, S), && (22)\\
\delta(CSeq(M_1, M_2) :: C, V, E, S) &= \delta(M_1 :: M_2 :: C, V, E, S). && (23)
\end{aligned}
$$

π lib declarations

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

$$
\begin{array}{lcl}
\langle \mathit{Statement} \rangle & ::= & \texttt{<Dec>} \\
\langle \mathit{Exp} \rangle & ::= & \text{'Ref'(<Exp>)>} \mid \text{'DeRef'(<Id>)} \mid \text{'ValRef'(<Id>)} \\
\langle \mathit{Dec} \rangle & ::= & \text{'Bind'(<Id>, <Exp>)} \mid \text{'DSeq'(<Dec>, <Dec>)} \\
\langle \mathit{Cmd} \rangle & ::= & \text{'Blk'(<Dec>, <Cmd>)}
\end{array}
$$

π automaton semantics for π lib 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 lib declarations II

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

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

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

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

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

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

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

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

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

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

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

π lib abstractions

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

  ⟨*Dec*⟩ ::= ‘Bind’(<Id>, <Abs>)

  ⟨*Abs*⟩ ::= ‘Abs’(<Formals>, <Blk>)

  ⟨*Formals*⟩ ::= ‘Formals‘(<Id>*)

  ⟨*Cmd*⟩ ::= ‘Call‘(<Id>, <Actuals>)

  ⟨*Actuals*⟩ ::= ‘Actuals‘(<Exp>*)

π automaton semantics for π lib 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.

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

π automaton semantics for π lib abstractions — Example I

```
Imπ source code:
# In this example we encapsulate the iterative calculation
# of the factorial within a function call.
let var z = 1
in
    let fn f(x) =
        let var y = x
        in
            while not (y == 0)
            do
                z := z * y
                y := y - 1
    in f(10)
```

π automaton semantics for π lib abstractions — Example II

```
π lib AST:
Blk(Bind(Id(z), Ref(Num(1))),
  Blk(BindAbs(Id(f), Abs(Id(x),
    Blk(Bind(Id(y), Ref(Id(x))),
      Loop(Not(Eq(Id(y), Num(0))),
        CSeq(Assign(Id(z), Mul(Id(z), Id(y))),
            Assign(Id(y), Sub(Id(y), Num(1)))))))),
    Call(Id(f), Num(10))))
```

π automaton semantics for π lib abstractions I

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

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

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

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

π lib 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 lib 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.

$$
\begin{aligned}
Rec &: Formals \times Blk \times Env \times Env \to Bindable \\
unfold &: Env \to Env \\
reclose_E &: Env \to Env
\end{aligned}
$$

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

$$
\begin{aligned}
\mathit{unfold}(E) &= \mathit{reclose}_E(E) && (38)\\
\mathit{reclose}_E(I \mapsto \mathit{Closure}(F,B,E')) &= (I \mapsto \mathit{Rec}(F,B,E',E)) && (39)\\
\mathit{reclose}_E(I \mapsto \mathit{Rec}(F,B,E',E'')) &= (I \mapsto \mathit{Rec}(F,B,E',E)) && (40)\\
\mathit{reclose}_E(I \mapsto v) &= (I \mapsto v) \text{ if } v \neq \mathit{Closure}(F,B,E) && (41)\\
\mathit{reclose}_E(E_1 \cup E_2) &= \mathit{reclose}_E(E_1) \cup \mathit{reclose}_E(E_2) && (42)\\
\mathit{reclose}_E(\emptyset) &= \emptyset && (43)
\end{aligned}
$$

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

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

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

π lib expressions in Python I

https://nbviewer.jupyter.org/github/ChristianoBraga/PiFramework/blob/master/python/pi.ipynb

```
class Statement:
    def __init__(self, *args):
        self.opr =args
    def __str__(self):
        ret =str(self.__class__.__name__)+"("
        for o in self.opr:
            ret +=str(o)
        ret +=")"
        return ret
class Exp(Statement): pass
class ArithExp(Exp): pass
```

π lib expressions in Python II

```
class Num(ArithExp):
    def __init__(self, f):
        assert(isinstance(f, int))
        ArithExp.__init__(self,f)
class Sum(ArithExp):
    def __init__(self, e1, e2):
        assert(isinstance(e1, Exp) and isinstance(e2, Exp))
        ArithExp.__init__(self, e1, e2)
...
```

π lib expressions in Python III

```
class BoolExp(Exp): pass
class Eq(BoolExp):
    def __init__(self, e1, e2):
        assert(isinstance(e1, Exp) and isinstance(e2, Exp))
        BoolExp.__init__(self, e1, e2)
...
```

π lib expressions in Python IV

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

Sum(Num(1)Mul(Num(2)Num(4)))
```

π lib expressions in Python V

```
exp2 =Mul(2, 1)
------------------------------------------------------------------------
---
AssertionError Traceback (most recent call last)
<ipython-input-7-00fd40a79a54> in <module>()
---->1 exp2 =Mul(2, 1)

<ipython-input-5-42a82e58862f> in __init__(self, e1, e2)
     28 class Mul(ArithExp):
     29 def __init__(self, e1, e2):
--->30 assert(isinstance(e1, Exp) and isinstance(e2, Exp))
     31 ArithExp.__init__(self, e1, e2)
     32 class BoolExp(Exp): pass

AssertionError:
```

π automaton for π lib 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 lib expressions II

```
class ExpPiAut(dict):
    def __init__(self):
        self["val"] =ValueStack()
        self["cnt"] =ControlStack()
    def __evalSum(self, e):
        e1 =e.opr[0]
        e2 =e.opr[1]
        self.pushCnt(ExpKW.SUM)
        self.pushCnt(e1)
        self.pushCnt(e2)
    def pushCnt(self, e):
        cnt =self.cnt()
        cnt.append(e)
...
```

π automaton for π lib expressions III

```
ea =ExpPiAut()
print(exp)
ea.pushCnt(exp)
while not ea.emptyCnt():
    ea.eval()
    print(ea)
```

π automaton for π lib expressions IV

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

π lib commands I

```
class Cmd(Statement): pass
class Id(Exp):
    def __init__(self, s):
        assert(isinstance(s, str))
        Exp.__init__(self, s)
class Assign(Cmd):
    def __init__(self, i, e):
        assert(isinstance(i, Id) and isinstance(e, Exp))
        Cmd.__init__(self, i, e)
class Loop(Cmd):
    def __init__(self, be, c):
        assert(isinstance(be, BoolExp) and isinstance(c, Cmd))
        Cmd.__init__(self, be, c)
class CSeq(Cmd):
    def __init__(self, c1, c2):
        assert(isinstance(c1, Cmd) and isinstance(c2, Cmd))
        Cmd.__init__(self, c1, c2)
```

π lib commands II

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

π automaton for π lib commands I

Environment, Location, Store and commands opcodes.

```
## Commands
class Env(dict): pass
class Loc(int): pass
class Sto(dict): pass
class CmdKW:
    ASSIGN ="#ASSIGN"
    LOOP ="#LOOP"
```

π automaton for π lib commands II

π automaton for commands extends the π automaton for expressions.

```
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 lib commands III

π semantics for assignment.

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

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

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

```python
    def __evalAssign(self, c):
        i =c.opr[0]
        e =c.opr[1]
        self.pushVal(i.opr[0])
        self.pushCnt(CmdKW.ASSIGN)
        self.pushCnt(e)
    def __evalAssignKW(self):
        v =self.popVal()
        i =self.popVal()
        l =self.getLoc(i)
        self.updateStore(l, v)
```

π automaton for π lib commands IV

π semantics for identifiers.

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

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

π automaton for π lib commands V

π semantics for loop: recursive step.

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

```
    def __evalLoop(self, c):
        be =c.opr[0]
        bl =c.opr[1]
        self.pushVal(Loop(be, bl))
        self.pushVal(bl)
        self.pushCnt(CmdKW.LOOP)
        self.pushCnt(be)
```

π automaton for π lib commands VI

π semantics for loop: basic steps.

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

```python
    def __evalLoopKW(self):
        t =self.popVal()
        if t:
            c =self.popVal()
            lo =self.popVal()
            self.pushCnt(lo)
            self.pushCnt(c)
        else:
            self.popVal()
            self.popVal()
```

π automaton for π lib commands VII

π semantics for command composition.

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

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

π automaton for π lib commands VIII

Commands are now on the top of the food chain.

```
    def eval(self):
        c =self.popCnt()
        if isinstance(c, Assign):
            self.__evalAssign(c)
        elif c ==CmdKW.ASSIGN:
            self.__evalAssignKW()
        elif isinstance(c, Id):
            self.__evalId(c.opr[0])
        elif isinstance(c, Loop):
            self.__evalLoop(c)
        elif c ==CmdKW.LOOP:
            self.__evalLoopKW()
        elif isinstance(c, CSeq):
            self.__evalCSeq(c)
        else:
            self.pushCnt(c)
            ExpPiAut.eval(self)
```

π lib declarations in Python I

DeRef and ValRef not implemented yet.

```
## Declarations
class Dec(Statement): pass
class Bind(Dec):
   def __init__(self, i, e):
       assert (isinstance(i, Id) and isinstance(e, Exp))
       Dec.__init__(self, i, e)
class Ref(Exp):
   def __init__(self, e):
       assert (isinstance(e, Exp))
       Exp.__init__(self, e)
class Cns(Exp):
   def __init__(self, e):
       assert (isinstance(e, Exp))
       Exp.__init__(self, e)
class Blk(Cmd):
   def __init__(self, d, c):
```

π lib declarations in Python II

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

π lib declarations in Python III

```
class DSeq(Dec):
    def __init__(self, d1, d2):
        assert (isinstance(d1, Dec) and isinstance(d2, Dec))
        Dec.__init__(self, d1, d2)
```

π lib declarations in Python IV

```
## Declarations
class DecExpKW(ExpKW):
    REF ="#REF"
    CNS ="#CNS"

class DecCmdKW(CmdKW):
    BLKDEC ="#BLKDEC"
    BLKCMD ="#BLKCMD"

class DecKW:
    BIND ="#BIND"
    DSEQ ="#DSEQ"

class DecPiAut(CmdPiAut):
    def __init__(self):
        self["locs"] =[]
        CmdPiAut.__init__(self)

```

π lib declarations in Python V

```
    def locs(self):
        return self["locs"]

    def pushLoc(self, l):
        ls =self.locs()
        ls.append(l)

    def __evalRef(self, e):
        ex =e.opr[0]
        self.pushCnt(DecExpKW.REF)
        self.pushCnt(ex)

    def __newLoc(self):
        sto =self.sto()
        if sto:
            return max(list(sto.keys())) +1
        else:
            return 0.0

```

π lib declarations in Python VI

```
    def __evalRefKW(self):
        v =self.popVal()
        l =self.__newLoc()
        self.updateStore(l, v)
        self.pushLoc(l)
        self.pushVal(l)

    def __evalBind(self, d):
        i =d.opr[0]
        e =d.opr[1]
        self.pushVal(i)
        self.pushCnt(DecKW.BIND)
        self.pushCnt(e)

    def __evalBindKW(self):
        l =self.popVal()
        i =self.popVal()
        x =i.opr[0]
        self.pushVal({x: l})
```

π lib declarations in Python VII

```
    def __evalDSeq(self, ds):
        d1 =ds.opr[0]
        d2 =ds.opr[1]
        self.pushCnt(DecKW.DSEQ)
        self.pushCnt(d2)
        self.pushCnt(d1)

    def __evalDSeqKW(self):
        d2 =self.popVal()
        d1 =self.popVal()
        d1.update(d2)
        self.pushVal(d1)

    def __evalBlk(self, d):
        ld =d.opr[0]
        c =d.opr[1]
        l =self.locs()
        self.pushVal(list(l))
```

π lib declarations in Python VIII

```
        self.pushVal(c)
        self.pushCnt(DecCmdKW.BLKDEC)
        self.pushCnt(ld)

    def __evalBlkDecKW(self):
        d =self.popVal()
        c =self.popVal()
        l =self.locs()
        self.pushVal(l)
        en =self.env()
        ne =en.copy()
        ne.update(d)
        self.pushVal(en)
        self["env"] =ne
        self.pushCnt(DecCmdKW.BLKCMD)
        self.pushCnt(c)

    def __evalBlkCmdKW(self):
        en =self.popVal()
```

π lib declarations in Python IX

```
        ls =self.popVal()
        self["env"] =en
        s =self.sto()
        s ={k: v for k, v in s.items() if k not in ls}
        self["sto"] =s
        # del ls
        ols =self.popVal()
        self["locs"] =ols

    def eval(self):
        d =self.popCnt()
        if isinstance(d, Bind):
            self.__evalBind(d)
        elif d ==DecKW.BIND:
            self.__evalBindKW()
        elif isinstance(d, DSeq):
            self.__evalDSeq(d)
        elif d ==DecKW.DSEQ:
            self.__evalDSeqKW()
```

π lib declarations in Python X

```
        elif isinstance(d, Ref):
            self.__evalRef(d)
        elif d ==DecExpKW.REF:
            self.__evalRefKW()
        elif isinstance(d, Blk):
            self.__evalBlk(d)
        elif d ==DecCmdKW.BLKDEC:
            self.__evalBlkDecKW()
        elif d ==DecCmdKW.BLKCMD:
            self.__evalBlkCmdKW()
        else:
            self.pushCnt(d)
            CmdPiAut.eval(self)
```

π lib declarations in Python XI

```
dc =DecPiAut()
fac =Loop(Not(Eq(Id("y"), Num(0))),
         CSeq(Assign(Id("x"), Mul(Id("x"), Id("y"))),
              Assign(Id("y"), Sub(Id("y"), Num(1)))))
dec =DSeq(Bind(Id("x"), Ref(Num(1))),
         Bind(Id("y"), Ref(Num(200))))
fac_blk =Blk(dec, fac)
dc.pushCnt(fac_blk)
while not dc.emptyCnt():
   aux =dc.copy()
   dc.eval()
   if dc.emptyCnt():
       print(aux)
```

π lib 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 lib abstractions in Python II

```python
        else:
            raise IllFormed(self, f)

    def formals(self):
        return self._opr[0]

    def blk(self):
        return self._opr[1]

    def __str__(self):
        ret =str(self.__class__.__name__) +"("
        formals =self.formals()
        ret +=str(formals[0]) # First formal argument
        for i in range(1, len(formals)):
            ret +=", "
            ret +=str(formals[i]) # Remaining formal arguments
        ret +=", "
        ret +=str(self.blk()) # Abstraction block
        ret +=")"
```

π lib abstractions in Python III

```python
        return ret

class BindAbs(Bind):
    '''
    BindAbs is a form of bind but that receives an Abs instead of an
    expression.
    '''
    def __init__(self, i, p):
        if isinstance(i, Id):
            if isinstance(p, Abs):
                Dec.__init__(self, i, p)
            else:
                raise IllFormed(self, p)
        else:
            raise IllFormed(self, i)

class Actuals(list):
    def __init__(self, a):
        if isinstance(a, list):
```

π lib abstractions in Python IV

```
            for e in a:
                if not isinstance(e, Exp):
                    raise IllFormed(self, e)
            self.append(a)
        else:
            raise IllFormed(self, a)

class Call(Cmd):
    def __init__(self, f, actuals):
        if isinstance(f, Id):
            if isinstance(actuals, list):
                Cmd.__init__(self, f, actuals)
            else:
                raise IllFormed(self, actuals)
        else:
            raise IllFormed(self, f)

    def caller(self):
        return self.operand(0)
```

π lib abstractions in Python V

```

    def actuals(self):
        return self.operand(1)

class Closure(dict):
    def __init__(self, f, b, e):
        if isinstance(f, list):
            if isinstance(b, Blk):
                # I wanted to write assert(isinstance(e, Env)) but it
                                                   fails.
                if isinstance(e, dict):
                    self["for"] =f # Formal parameters
                    self["env"] =e # Current environment
                    self["block"] =b # Procedure block
                else:
                    raise IllFormed(self, e)
            else:
                raise IllFormed(self, b)
        else:
```

π lib abstractions in Python VI

```python
            raise IllFormed(self, f)

    def __str__(self):
        ret =str(self.__class__.__name__) +"("
        formals =self.formals()
        fst_formal =formals[0] # First formal argument
        ret +=str(fst_formal)
        for i in range(1, len(formals)):
            ret +=", "
            formal =formals[i] # Remaining formal arguments
            ret +=str(formal)
        ret +=", "
        ret +=str(self.blk()) # Closure block
        ret +=")"
        return ret

    def formals(self):
        return self['for']

```

π lib abstractions in Python VII

```
    def env(self):
        return self['env']

    def blk(self):
        return self['block']

class AbsPiAut(DecPiAut):
    def __evalAbs(self, a):
        if not isinstance(a, Abs): # p must be an abstraction
            raise EvaluationError(self, "Function __evalAbs called with
                                                    no abstraction but with "
                                                    , a, " instead.")
        else:
            f =a.formals() # Formal parameters
            b =a.blk() # Body
            e =self.env() # Current environment
            # Closes the given abs. with the current env
            c =Closure(f, b, e)
            # Closure c is pushed to the value stack such that
```

π lib abstractions in Python VIII

```python
            self.pushVal(c)
            # a BIND may create a new binding to a given identifier.

    def __match(self, f, a):
        '''
        Given a list of formal parameters and a list of actual parameters,
        it returns an environment relating the elements of the former with the latter.
        '''
        if isinstance(f, list):
            if isinstance(a, list):
                if len(f) ==0:
                    return {}
                if len(f) ==len(a) and len(f) >0:
                # For some reason, f[0] is a tuple, not an Id.
                    f0 =f[0]
                    a0 =a[0]
                    b0 ={f0.id(): a0.num()}
```

π lib abstractions in Python IX

```
                if len(f) ==1:
                    return b0
                else:
                    # For some reason, f[0] is a tuple, not an Id.
                    f1 =f[1]
                    a1 =a[1]
                    b1 ={f1.id(): a1.num()}
                    e =b0.update(b1)
                    for i in range(2, len(f)):
                        fi =f[i][0]
                        ai =a[i][0]
                        e.update({fi.id(): ai.num()})
                    return e
            else:
                raise EvaluationError("Call to '__match' on " +str(self) +
                                      ": " +"formals and
                                      actuals differ in
                                      size.")
        else:
```

π lib abstractions in Python X

```
            raise EvaluationError("Call to '__match' on " +str(self) +":
                                                    " +" no formals, but with
                                                     ", f, " instead.")

    def __evalCall(self, c):
        '''
        Essentially, a call is translated into a block.
        If we were progrmming pi in a symbolic language,
        we could simply crete a proper block and push it to the control
                                                 stack.
        However, the environment is not symbolic: is a dictionary of
                                                 objects.
        To create a block we would need to "pi-lib-fy" it, that is,
                                                 recreate the
        pi lib tree from the concrete environmnet and joint it with
                                                 matches created
        also at pi lib level. These would be pushed back into the control
                                                  stack and
```

π lib abstractions in Python XI

```
        reobjectifyed. Thus, to avoid pi-libfication and reevaluatuation of the
        environment we manipulate it at the object level, which is dangerous but
        seems to be correct.

        In this implementation, actual parameters are already evaluated.
        '''
        if not isinstance(c, Call): # c must be a Call object
            raise EvaluationError("Call to __evalCall with no Call object but with ", c, " instead.")
        else:
            # Procedure to be called
            caller =c.caller()
            # Retrieves the current environment.
            e =self.env()
            # Retrieves the closure associated with the caller function.
            clos =e[caller.id()]
```

π lib abstractions in Python XII

```
        # Retrieves the actual parameters from the call.
        a =c.actuals()
        # Retrieves the formal parameters from the closure.
        f =clos.formals()
        # Matches formals and actuals, creating an environment.
        d =self.__match(f, a)
        # Retrives the closure's environment.
        ce =clos.env()
        # The caller's block must run on the closures environment
        # overwritten with the matches.
        d.update(ce)
        self["env"] =d
        self.pushVal(self.locs())
        # Saves the current environment in the value stack.
        self.pushVal(e)
        # Pushes the keyword BLKCMD for block completion.
        self.pushCnt(DecCmdKW.BLKCMD)
        # Pushes the body of the caller function into the control
                                         stack.
```

π lib abstractions in Python XIII

```
            self.pushCnt(clos.blk())

    def eval(self):
        d =self.popCnt()
        if isinstance(d, Abs):
            self.__evalAbs(d)
        elif isinstance(d, Call):
            self.__evalCall(d)
        else:
            self.pushCnt(d)
            DecPiAut.eval(self)
```

Complete Imp grammar in EBNF notation I

```
@@grammar::IMP
@@eol_comments ::/#.*?/start = @ : cmd ;



cmd =nop | let | assign | loop | call ;

call =i:identifier '(' { a:actual }* ')' ;

actual =e1:expression { ',' e2:expression }* | {} ;

nop ='nop' ;

loop =op:'while' ~ e:expression 'do' { c:cmd }+ ;

assign =id:identifier op:':=' ~ e:expression ;

let =op:'let' ~ d:dec 'in' { c:cmd }+ ;
```

Complete Imp grammar in EBNF notation II

```
dec =var | fn ;

var =op:'var' ~ id:identifier '=' e:expression ;

fn =op:'fn' ~ id:identifier '(' f:formal ')' '=' c:cmd ;

formal =i1:identifier { ',' i2:identifier }* | {} ;

expression =@:bool_expression ;

bool_expression =negation | equality | conjunction | disjunction
               | lowereq | greatereq | lowerthan | greaterthan
               | add_expression ;

equality =left:add_expression op:"==" ~ right:bool_expression ;

conjunction =left:add_expression op:"and" ~ right:bool_expression ;
```

Complete Imp grammar in EBNF notation III

```
disjunction =left:add_expression op:"or" ~ right:bool_expression ;

lowereq =left:add_expression op:"<=" ~ right:add_expression ;

greatereq =left:add_expression op:">=" ~ right:add_expression ;

lowerthan =left:add_expression op:"<" ~ right:add_expression ;

greaterthan =left:add_expression op:">" ~ right:add_expression ;

parentesisexp ='(' ~ @:bool_expression ')' ;

negation =op:'not' ~ b:bool_expression ;

add_expression =addition | subtraction | @:mult_expression ;

addition =left:mult_expression op:"+" ~ right:add_expression ;

subtraction =left:mult_expression op:"-" ~ right:add_expression ;
```

Complete Imp grammar in EBNF notation IV

```
mult_expression =multiplication | division
                | atom
                | parentesisexp ;

multiplication =left:atom op:"*" ~ right:mult_expression ;

division =left:atom op:"/" ~ right:mult_expression ;

atom =number | truth | identifier ;

number =/\d+/ ;

identifier =/(?!\d)\w+/ ;

truth ='True' | 'False' ;
```

Example: iterative factorial

```
# The classic iterative factorial example
let var z = 1
in
    let var y = 10
    in
        while not (y == 0)
        do
            z := z * y
            y := y - 1
```

Example: iterative factorial within a function

```
# In this example we encapsulate the iterative calculation
# of the factorial within a function call.
let var z = 1
in
    let fn f(x) =
        let var y = x
        in
            while not (y == 0)
            do
                z := z * y
                y := y - 1
    in f(10)
```