Notes on Formal Compiler Construction with the Π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

March 26, 2021

http://github.com/ChristianoBraga/PiFramework

Instituto de computação uff

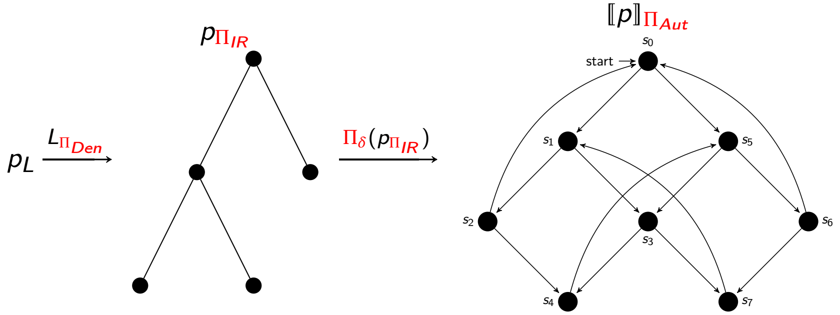
Compiler pipeline

$$p_L \xrightarrow{lexical\ \ analysis} tokens_p \xrightarrow{syntactical\ \ analysis} ast_p \xrightarrow{semantic\ \ analysis} ast_p\checkmark \xrightarrow{code\ \ generation} p_{asm}$$

Compiler pipeline and formal languages

$$
\begin{array}{l}
p_L \xrightarrow[]{\substack{\textit{lexical analysis}\\ \textit{Linear Grm.}}} tokens_p \\
\xrightarrow[]{\substack{\textit{syntactical analysis}\\ \textit{Context-free Grm.}}} ast_p \\
\xrightarrow[]{\substack{\textit{semantic analysis}\\ \textit{Context-sensitive Grm.}}} ast_p\checkmark \\
\xrightarrow[]{\substack{\textit{code generation}\\ \textit{Context-sens. Grm.}}} p_{asm} \\
\xrightarrow[]{\textit{code optimization}} p'_{asm}
\end{array}
$$

Compiler pipeline with the Π Framework I



Compiler pipeline with the Π Framework II

$$
\underset{code}{source} \;\; \overbrace{\xrightarrow{lexer} \circ \xrightarrow{parser} \circ \xrightarrow{\overset{AST}{transformer}} \circ \xrightarrow{\overset{type}{checker}} \circ \xrightarrow{\overset{code}{generator}}}^{Chomsky's\ hierarchy} \;\; \Pi\ IR \;\; \overbrace{\begin{array}{c} \xrightarrow{interpreter} \\ \xrightarrow{code\ generator} \\ \xrightarrow{model\ checker(P)} \\ \text{where } P \text{ is a property} \\ \text{in a suitable logic.} \end{array}}^{\Pi\ Automata} \;\; \begin{array}{c} output \\ machine\ code \\ counter\text{-}examples \end{array}
$$

- 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.

Compiler pipeline with the Π Framework III

- 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

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

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

$$\langle \mathit{boolean} \rangle ::= \text{‘}\mathtt{true}\text{’} \mid \text{‘}\mathtt{false}\text{’}$$

A calculator: concrete syntax

$$
\begin{array}{lcl}
\langle \mathit{exp} \rangle & ::= & \langle \mathit{aexp} \rangle \mid \langle \mathit{bexp} \rangle \\
\langle \mathit{aexp} \rangle & ::= & \langle \mathit{aexp} \rangle \text{ `+' } \langle \mathit{term} \rangle \mid \langle \mathit{aexp} \rangle \text{ `-' } \langle \mathit{term} \rangle \mid \langle \mathit{term} \rangle \\
\langle \mathit{term} \rangle & ::= & \langle \mathit{term} \rangle \text{ `}*\text{' } \langle \mathit{factor} \rangle \mid \langle \mathit{term} \rangle \text{ `/' } \langle \mathit{factor} \rangle \mid \langle \mathit{factor} \rangle \\
\langle \mathit{factor} \rangle & ::= & \text{`(' } \langle \mathit{aexp} \rangle \text{ `)' } \mid \langle \mathit{digits} \rangle \\
\langle \mathit{bexp} \rangle & ::= & \langle \mathit{boolean} \rangle \mid \text{`\textasciitilde' } \langle \mathit{bexp} \rangle \mid \langle \mathit{bexp} \rangle \langle \mathit{boolop} \rangle \langle \mathit{bexp} \rangle \\
 & \mid & \langle \mathit{aexp} \rangle \langle \mathit{iop} \rangle \langle \mathit{aexp} \rangle \\
\langle \mathit{boolop} \rangle & ::= & \text{`=' } \mid \text{`/}\backslash\text{' } \mid \text{`}\backslash\text{/'} \\
\langle \mathit{iop} \rangle & ::= & \text{`<' } \mid \text{ `>' } \mid \text{ `<=' } \mid \text{ `>='}
\end{array}
$$

A calculator: abstract syntax

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

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

A calculator: Π denotations I

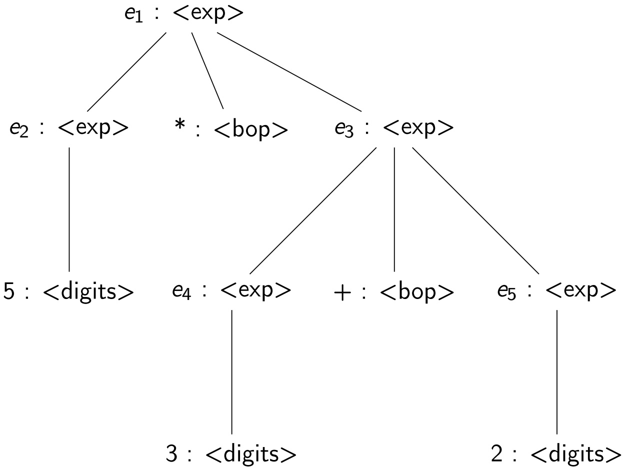
Let D in <digits>, B in <boolean> and E_1, E_2 in <exp>,

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

A calculator: Π denotations II

- Π denotations are *functions* $[\![\cdot]\!]_\Pi : AST \to \Pi\ IR$, where AST denotes the *datatype* for the abstract syntax tree and $\Pi\ IR$ denotes the datatype for $\Pi\ IR$ 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

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

A calculator: executing Π IR with Π automata I

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 $[s_1, s_2, \ldots, s_n]$.)

A calculator: executing Π IR with Π automata II

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

Excerpt of Π IR expressions

$$
\begin{array}{lcl}
\langle \mathit{Statement} \rangle & ::= & \text{<Exp>} \\
\langle \mathit{Exp} \rangle & ::= & \text{<ArithExp>} \mid \text{<BoolExp>}\ \text{<Exp>} \\
\langle \mathit{ArithExp} \rangle & ::= & \text{`Num'(<digits>)} \mid \text{`Sum'(<Exp> , <Exp>)} \mid \\
 & & \text{`Sub'(<Exp>, <Exp>)} \mid \text{`Mul'(<Exp>, <Exp>)} \\
\langle \mathit{BoolExp} \rangle & ::= & \text{`Eq'(<Exp>, <Exp>)} \mid \text{`Not'(<Exp>)}
\end{array}
$$

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.

$\langle \mathit{Statement} \rangle$ ::= <Cmd>

$\langle \mathit{Exp} \rangle$:= 'Id'(<String>)

$\langle \mathit{Cmd} \rangle$ ::= '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.
- $Store = Loc \mapsto Storable$, $Env = Id \mapsto Bindable$, $Loc \subseteq Storable$, $\mathbb{N} \subseteq Loc, Bindable$.
- 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 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,

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

Π 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.

⟨*Statement*⟩ ::= <Dec>

⟨*Exp*⟩ ::= ‘Ref’(<Exp>)> | ‘DeRef’(<Id>) | ‘ValRef’(<Id>)

⟨*Dec*⟩ ::= ‘Bind’(<Id>, <Exp>) | ‘DSeq’(<Dec>, <Dec>)

⟨*Cmd*⟩ ::= ‘Blk’(<Dec>, <Cmd>)

Π automaton semantics for Π IR declarations I

Let $BlockLocs = \mathscr{P}(Loc)$, 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

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

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

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

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

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

$$\delta(\#BIND :: C, B :: W :: V, E, S, L) = \delta(C, [W \mapsto B] :: V, E, S, L), \tag{29}$$

$$\delta(DSeq(D_1, D_2), X) :: C, V, E, S, L) = \delta(D_1 :: D_2 :: \#DSEQ :: C, V, E, S, L), \tag{30}$$

$$\delta(\#DSEQ :: C, E :: E' :: H :: V, E, S, L) = \delta(C, (E \cup E') :: H :: V, E, S, L), \textbf{where } E' \in \textbf{\textit{Env}}, H \notin \textbf{\textit{Env}}, \tag{31}$$

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

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

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

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.

$$\begin{array}{lcl}
\langle \mathit{Dec} \rangle & ::= & \text{`Bind'}(\text{<Id>}, \text{<Abs>}) \\
\langle \mathit{Abs} \rangle & ::= & \text{`Abs'}(\text{<Formals>}, \text{<Blk>}) \\
\langle \mathit{Formals} \rangle & ::= & \text{<Id>}^{*} \\
\langle \mathit{Cmd} \rangle & ::= & \text{`Call`}(\text{<Id>}, \text{<Actuals>}) \\
\langle \mathit{Actuals} \rangle & ::= & \text{<Exp>}^{*}
\end{array}$$

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

```
# Iterative factorial
def fat (x) {
    var z = x, y = 1 ;
    while (z > 0)
    {
        y := y * z ;
        z := z − 1 ;
        print(y)
    }
}
fat(2)
```

Π automaton semantics for Π IR abstractions — Example II

```
Π IR syntax tree:
Blk(
 BindAbs(Id(fat), Abs([Id(x)],
  Blk(Bind(Id(z), Ref(Id(x))),
    Blk(Bind(Id(y), Ref(1)),
       Loop(Gt(Id(z), 0),
          Blk(CSeq(CSeq(Assign(Id(y), Mul(Id(y), Id(z))), Assign(Id(z), Sub(Id(z), 1))),
                   Print(Id(y)))))))))),
 Call(Id(fat), [2]))
```

Π automaton semantics for Π IR abstractions I

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

$$\delta(Abs(F,B) :: C, V, E, S, L) = \delta(C, Closure(F,B,E) :: V, E, S, L) \quad (35)$$

$$\delta(Call(I,[X_1,X_2,\ldots,X_n])) :: C, V, E, S, L) = \quad (36)$$
$$\delta(X_n :: X_{n-1} :: \ldots :: X_1 :: \#CALL(I,n) :: C, V, E, S, L)$$

$$\delta(\#CALL(I,n) :: C, (V_1 :: V_2 :: \ldots V_n :: V), E, S, L) = \quad (37)$$
$$\delta(B :: \#BLKCMD :: C, E :: V, E', S, L)$$
$$\textbf{where } E = \{I \mapsto Closure(F,B,E_1)\} \cup E_2,$$
$$E' = E_1 / match(F, [V_1, V_2, \ldots, V_n])$$

Π automaton semantics for Π IR abstractions II

$$match : Id^* \times Values^* \rightarrow Env$$
$$match(fl, al) = \text{if} |fl| \neq |al| \text{ than } \{\} \text{ else } \_match(fl, al, \{\})$$

$$\_match : Id^* \times Values^* \times Env \rightarrow Env$$
$$\_match([], [], E) = E$$
$$\_match(f, a, E) = \{f \mapsto a\} \cup E$$
$$\_match(f :: fl, a :: al, E) = \_match(fl, al, \{f \mapsto a\} \cup E)$$

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.

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

Π automaton semantics for Π 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

$$
\begin{aligned}
&\delta(Rbnd(I, Abs(F, B)) :: C, V, E, S, L) = \\
&\qquad \delta(C, unfold(I \mapsto Closure(F, B, E)) :: V, E, S, L) \qquad (44) \\
&\delta(\#CALL(I, n) :: C, V_1 :: V_2 :: \ldots :: V_n :: V, E, S, L) = \\
&\qquad \delta(B :: \#BLKCMD :: C, E :: L :: V, E', S, \emptyset) \qquad (45) \\
&\textbf{where } E = \{I \mapsto Rec(F, B, E_1, E_2)\} \cup E_3, \\
&\qquad E' = E_1 / unfold(E_2) / match(F, [V_1, V_2, \ldots, V_n])
\end{aligned}
$$