Notes on Formal Compiler Construction with the Π Framework

Π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

August 11, 2021

http://github.com/ChristianoBraga/PiFramework

Compiler pipeline

$$p_L \xrightarrow{\text{lexical analysis}} tokens_p$$
$$\xrightarrow{\text{syntactical analysis}} ast_p$$
$$\xrightarrow{\text{semantic analysis}} ast_p\checkmark$$
$$\xrightarrow{\text{code generation}} p_{asm}$$
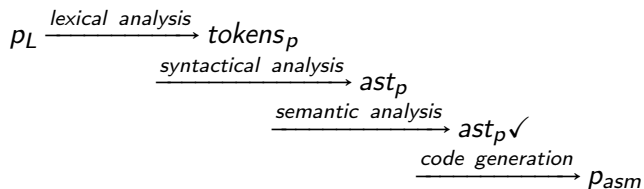
Compiler pipeline and formal languages

$$p_L \xrightarrow[\text{Linear Grm.}]{\text{lexical analysis}} tokens_p$$

$$\xrightarrow[\text{Context-free Grm.}]{\text{syntactical analysis}} ast_p$$

$$\xrightarrow[\text{Context-sensitive Grm.}]{\text{semantic analysis}} ast_p\checkmark$$

$$\xrightarrow[\text{Context-sens. Grm.}]{\text{code generation}} p_{asm}$$

$$\xrightarrow{\text{code optimization}} p'_{asm}$$
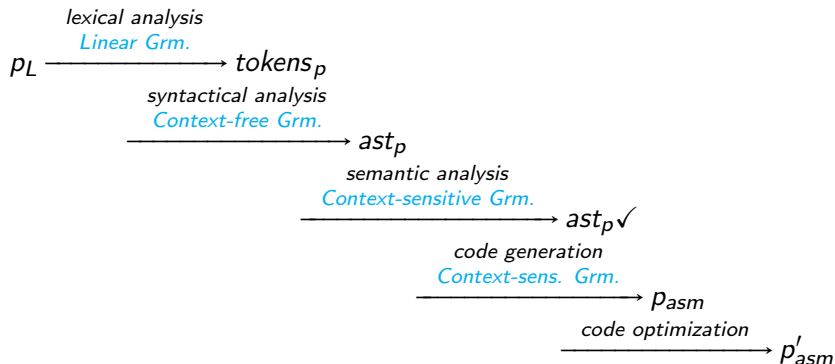
Compiler pipeline with the Π Framework I

Compiler pipeline with the Π Framework II

$$\Pi$$
$$Automata$$

$$\textit{Chomsky's hierarchy}$$

$$\xrightarrow{\textit{interpreter}} \quad output$$

$$source \xrightarrow{lexer} \circ \xrightarrow{parser} \circ \xrightarrow[\textit{transformer}]{\textit{AST}} \circ \xrightarrow[\textit{checker}]{\textit{type}} \circ \xrightarrow[\textit{generator}]{\textit{code}} \Pi \; IR \xrightarrow{\textit{code generator}} machine\ code$$

$$code \xrightarrow{\textit{model checker}(P)} counter\text{-}examples$$

where P is a property

in a suitable logic.
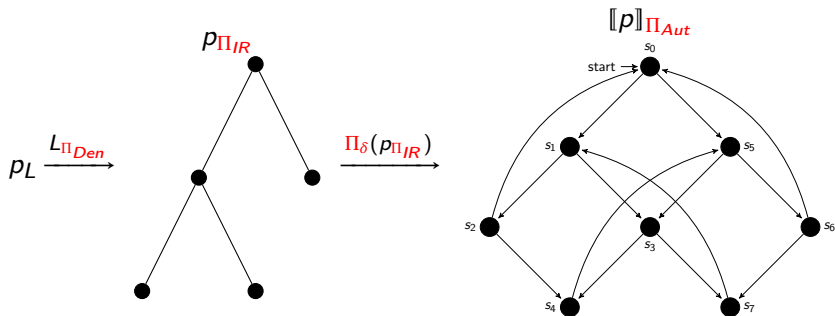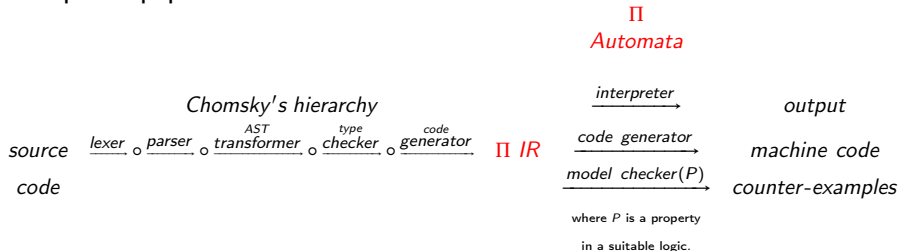
- Π IR defines a set of constructions common to many programming languages.
- Π IR constructions have a formal automata-based semantics in Π automata.
- One may execute (or validate) a program in a given language by running its associated Π IR program.

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

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

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

$\langle \textit{boolean} \rangle \qquad ::= \text{'true'} \mid \text{'false'}$

A calculator: concrete syntax

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

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

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

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

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

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

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

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 Π *IR* denotes the datatype for Π *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

e_1 : <exp>

e_2 : <exp>   * : <bop> e_3 : <exp>

5 : <digits> e_4 : <exp>   + : <bop> e_5 : <exp>

3 : <digits>   2 : <digits>
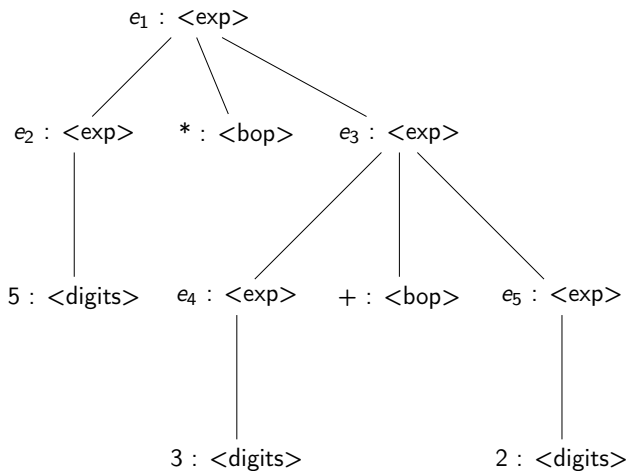
A calculator: Π denotations IV

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

$$Mul(\llbracket 5 \rrbracket_\Pi, \llbracket (3+2) \rrbracket_\Pi) = Mul(Num(5), \llbracket (3+2) \rrbracket_\Pi) \qquad \textit{by Equation 1}$$

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

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

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

Instituto de
Computação

A calculator: executing Π IR with Π automata I

A Π automaton is a 5-tuple $\mathcal{A} = (G, Q, \delta, q_0, F)$, 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

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

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

$$\delta([Mul(Num(5), Sum(Num(3), Num(2))], \emptyset, \emptyset) = \delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset)$$
$$\delta([Num(5), Sum(Num(3), Num(2)), \#MUL], \emptyset, \emptyset) = \delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset)$$
$$\delta([Sum(Num(3), Num(2)), \#MUL], [Num(5)], \emptyset) = \delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset)$$
$$\delta([Num(3), Num(2), \#SUM, \#MUL], [Num(5)], \emptyset) = \delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset)$$
$$\delta([Num(2), \#SUM, \#MUL], [Num(3), Num(5)], \emptyset) = \delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset)$$
$$\delta([\#SUM, \#MUL], [Num(2), Num(3), Num(5)], \emptyset) = \delta([\#MUL], [Num(5), Num(5)], \emptyset)$$
$$\delta([\#MUL], [Num(5), Num(5)], \emptyset) = \delta(\emptyset, [Num(25)], \emptyset)$$
$$\delta(\emptyset, [Num(25)], \emptyset) = Num(25)$$

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 $N, N_i \in \mathbb{N}$, $C, V \in L(G)^*$, $S \in Store$,

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

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

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

$$\dots$$

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

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

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

- 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.
- $Store = Loc \hookrightarrow Storable$, $Env = Id \hookrightarrow Bindable$, $Loc \subseteq Storable$, $\mathbb{N} \subseteq Loc, Bindable$.
- 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,

$$\delta(Id(W) :: C, V, E, S) = \delta(C, B :: V, E, S), \tag{17}$$
$$\textbf{where } E[W] = I \textbf{ and } S[I] = B,$$

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

$$\delta(\#ASSIGN :: C, T :: W :: V, E, S) = \delta(C, V, E, S'), \tag{19}$$
$$\textbf{where } E[W] = I \textbf{ and } S' = S/[I \mapsto T],$$

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

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

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

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

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>)

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

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

Π automaton semantics for Π IR declarations

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,

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

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

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

$$\delta(Bind(Id(W), X) :: C, V, E, S, L) = \delta(X :: \#BIND :: C, W :: V, E, S, L), \tag{27}$$

$$\delta(\#BIND :: C, B :: W :: V, E, S, L) = \delta(C, [W \mapsto B] :: V, E, S, L), \tag{28}$$

$$\delta(DSeq(D_1, D_2), X) :: C, V, E, S, L) = \delta(D_1 :: D_2 :: \#DSEQ :: C, V, E, S, L), \tag{29}$$

$$\delta(\#DSEQ :: C, E :: E' :: H :: V, E, S, L) = \delta(C, (E \cup E') :: H :: V, E, S, L), \textbf{where } E' \in \textbf{\textit{Env}}, H \notin \textbf{\textit{Env}}, \tag{30}$$

$$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: C, M :: L :: V, E, S, \varnothing), \tag{31}$$

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

$$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L), \textbf{ where } S' = S - L'. \tag{33}$$

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  # Iterative factorial
 2  def fat (x) {
 3      var z = x, y = 1 ;
 4      while (z > 0)
 5      {
 6          y := y * z ;
 7          z := z − 1 ;
 8          print(y)
 9      }
10  }
11  fat(2)
```

Π automaton semantics for Π IR abstractions — Example II

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

Π automaton semantics for Π IR abstractions I

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

$$\delta(\textit{Abs}(F,B) :: C, V, E, S, L) = \delta(C, \textit{Closure}(F,B,E) :: V, E, S, L) \qquad (34)$$

$$\delta(\textit{Call}(I,[X_1, X_2, \ldots, X_n])) :: C, V, E, S, L) = \qquad (35)$$
$$\delta(X_n :: X_{n-1} :: \ldots :: X_1 :: \#\textit{CALL}(I,n) :: C, V, E, S, L)$$

$$\delta(\#\textit{CALL}(I,n) :: C, (V_1 :: V_2 :: \ldots V_n :: V), E, S, L) = \qquad (36)$$
$$\delta(B :: \#\textit{BLKCMD} :: C, E :: V, E', S, L)$$
$$\textbf{where } E = \{I \mapsto \textit{Closure}(F,B,E_1)\} \cup E_2,$$
$$E' = E_1 / \textit{match}(F, [V_1, V_2, \ldots, V_n])$$

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.

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

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

$$unfold(E) = reclose_E(E) \tag{37}$$

$$reclose_E(I \mapsto Closure(F, B, E')) = (I \mapsto Rec(F, B, E', E)) \tag{38}$$

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

$$reclose_E(I \mapsto v) = (I \mapsto v) \text{ if } v \neq Closure(F, B, E) \tag{40}$$

$$reclose_E(E_1 \cup E_2) = reclose_E(E_1) \cup reclose_E(E_2) \tag{41}$$

$$reclose_E(\emptyset) = \emptyset \tag{42}$$

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

$$\delta(Rbnd(I, Abs(F, B)) :: C, V, E, S, L) =$$
$$\delta(C, unfold(I \mapsto Closure(F, B, E)) :: V, E, S, L) \quad (43)$$
$$\delta(\#CALL(I, n) :: C, V_1 :: V_2 :: \ldots :: V_n :: V, E, S, L) =$$
$$\delta(B :: \#BLKCMD :: C, E :: L :: V, E', S, \emptyset) \quad (44)$$
$$\textbf{where} \ \ E = \{I \mapsto Rec(F, B, E_1, E_2)\} \cup E_3,$$
$$E' = E_1 / unfold(E_2) / match(F, [V_1, V_2, \ldots, V_n])$$