Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

August 21, 2018

http://github.com/ChristianoBraga/BPLC

Compiler pipeline

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

Compiler pipeline and formal languages

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

Compiler pipeline with the π Framework

π *Automata*

Chomsky's hierarchy

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

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

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

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

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/BPLC`
- Notes on Formal Compiler Construction with the π Framework: `https://github.com/ChristianoBraga/BPLC/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

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

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>,

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

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

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

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

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

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

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

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

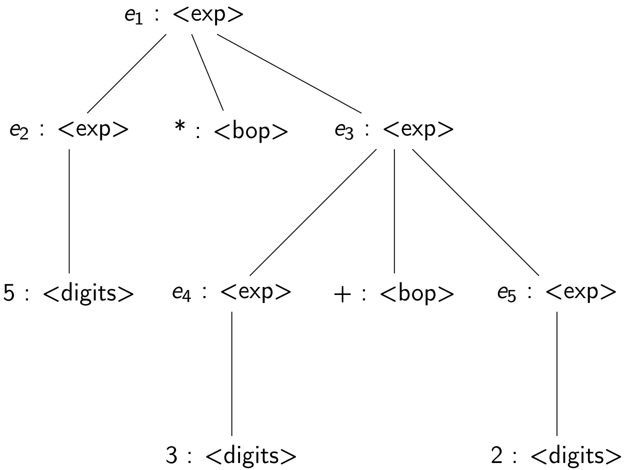
$$[\![E_1 > E_2]\!]_\pi = Gt([\![E_1]\!]_\pi, [\![E_2]\!]_\pi) \tag{9}$$

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

A calculator: π denotations II

- π denotations are *functions* $[\![\cdot]\!]_\pi : AST \rightarrow \pi\ lib$, 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 automata semantics for π lib expressions

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

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

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

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

$$\dots$$

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

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

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

- 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*.

Instituto de Computação

π lib commands

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

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

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

  $\langle \textit{Cmd} \rangle$ ::= '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 automata 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 $W \in String$, $l \in Loc$, $N, N_i \in \mathbb{N}$, $X \in < Exp >$, $M, M_1, M_2 \in < Cmd >, C, V \in L(G)^*$, $S \in Store$, $E \in Env$, and expression

π automata semantics for π lib commands II

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

$$\begin{aligned}
\delta(Id(W) :: C, V, E, S) &= \delta(C, Num(N) :: V, E, S), && (17)\\
&\quad \textbf{where } E[W] = I \textbf{ and } S[I] = N, \\
\delta(Assign(W, X) :: C, V, E, S) &= \delta(X :: \#ASSIGN :: C, W :: V, E, S'), && (18)\\
\delta(\#ASSIGN :: C, W :: V, E, S) &= \delta(C, V, E, S'), && (19)\\
&\quad \textbf{where } E[W] = I \textbf{ and } S' = S/[I \mapsto N], \\
\delta(Loop(X, M) :: C, V, E, S) &= \delta(X :: \#LOOP :: C, Loop(X, M) :: V, E, S), && (20)\\
\delta(\#LOOP :: C, Boo(true) :: Loop(X, M) :: V, E, S) &= \delta(M :: Loop(X, M) :: C, V, E, S), && (21)\\
\delta(\#LOOP :: C, Boo(false) :: Loop(X, M) :: V, E, S) &= \delta(C, V, E, S), && (22)\\
\delta(CSeq(M_1, M_2) :: C, V, E, S) &= \delta(M_1 :: M_2 :: C, V, E, S). && (23)
\end{aligned}$$

π lib expressions in Python I

https://github.com/ChristianoBraga/BPLC/blob/master/python/pi.ipynb

```python
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

```
## 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)
...
```

Instituto de Computação UFF

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.

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

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

$$\text{where } E[W] = l \text{ and } S' = S/[l \mapsto N].$$

```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.

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

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

Instituto de Computação

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.

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

```
    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)
```