Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação,
Universidade Federal Fluminense, Niterói, Brazil

August 14, 2018

http://github.com/ChristianoBraga/BPLC

Instituto de computação uff

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

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

Compiler architecture with the π Framework

$$\overbrace{\textit{source code} \xrightarrow{lexer} \circ \xrightarrow{parser} \circ \xrightarrow{\overset{AST}{transformer}} \circ \xrightarrow{\overset{type}{checker}} \circ \xrightarrow{\overset{code}{generator}}}^{Chomsky's\ hierarchy} \overbrace{\pi\ lib}^{\substack{\pi \\ Automata}} \begin{cases} \xrightarrow{interpreter} & output \\ \xrightarrow{code\ generator} & machine\ code \\ \xrightarrow[\text{where } P \text{ is a property in a suitable logic.}]{model\ checker(P)} & counter\text{-}examples \end{cases}$$

- 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

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

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

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

A calculator: concrete syntax

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

A calculator: abstract syntax

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

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

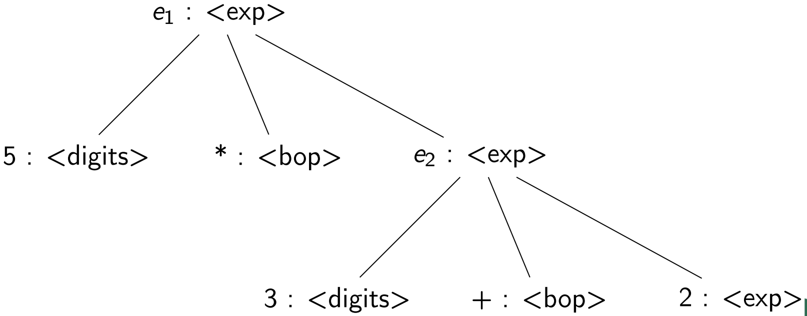
A calculator: π denotations I

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

$$\llbracket D \rrbracket_\pi = num(D) \tag{1}$$
$$\llbracket B \rrbracket_\pi = boo(B) \tag{2}$$
$$\llbracket E_1 + E_2 \rrbracket_\pi = add(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{3}$$
$$\llbracket E_1 - E_2 \rrbracket_\pi = sub(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{4}$$
$$\llbracket E_1 * E_2 \rrbracket_\pi = mul(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{5}$$
$$\llbracket E_1 / E_2 \rrbracket_\pi = div(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{6}$$
$$\llbracket E_1 < E_2 \rrbracket_\pi = lt(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{7}$$
$$\llbracket E_1 <= E_2 \rrbracket_\pi = le(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{8}$$
$$\llbracket E_1 > E_2 \rrbracket_\pi = gt(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{9}$$
$$\llbracket E_1 >= E_2 \rrbracket_\pi = ge(\llbracket E_1 \rrbracket_\pi, \llbracket E_2 \rrbracket_\pi) \tag{10}$$

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 π *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

$$
\begin{aligned}
[\![5 * (3+2)]\!]_\pi &= \mathit{mul}([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) && \text{by Equation 5} \\
\mathit{mul}([\![5]\!]_\pi, [\![(3+2)]\!]_\pi) &= \mathit{mul}(\mathit{num}(5), [\![(3+2)]\!]_\pi) && \text{by Equations 1} \\
\mathit{mul}(\mathit{num}(5), [\![(3+2)]\!]_\pi) &= \mathit{mul}(\mathit{num}(5), \mathit{add}([\![3]\!]_\pi, [\![2]\!]_\pi) && \text{by Equation 3} \\
\mathit{mul}(\mathit{num}(5), \mathit{add}([\![3]\!]_\pi, [\![2]\!]_\pi) &= \mathit{mul}(\mathit{num}(5), \mathit{add}(\mathit{num}(3), [\![2]\!]_\pi) && \text{by Equation 1} \\
\mathit{mul}(\mathit{num}(5), \mathit{add}(\mathit{num}(3), [\![2]\!]_\pi) &= \mathit{mul}(\mathit{num}(5), \mathit{add}(\mathit{num}(3), \mathit{num}(2)) && \text{by Equation 1}
\end{aligned}
$$

Instituto de Computação

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

$$\begin{aligned}
\delta([mul(num(5), add(num(3), num(2)], \emptyset, \emptyset) &= \delta([num(5), add(num(3), num(2)), \#MUL], \emptyset, \emptyset)\\
\delta([num(5), add(num(3), num(2)), \#MUL], \emptyset, \emptyset) &= \delta([add(num(3), num(2)), \#MUL], [num(5)], \emptyset)\\
\delta([add(num(3), num(2)), \#MUL], [num(5)], \emptyset) &= \delta([num(3), num(2), \#SUM, \#MUL], [num(5)], \emptyset)\\
\delta([num(3), num(2), \#SUM, \#MUL], [num(5)], \emptyset) &= \delta([num(2), \#SUM, \#MUL], [num(3), num(5)], \emptyset)\\
\delta([num(2), \#SUM, \#MUL], [num(3), num(5)], \emptyset) &= \delta([\#SUM, \#MUL], [num(2), num(3), num(5)], \emptyset)\\
\delta([\#SUM, \#MUL], [num(2), num(3), num(5)], \emptyset) &= \delta([\#MUL], [num(5), num(5)], \emptyset)\\
\delta([\#MUL], [num(5), num(5)], \emptyset) &= \delta(\emptyset, [num(25)], \emptyset)\\
\delta(\emptyset, [num(25)], \emptyset) &= num(25)
\end{aligned}$$