PL0 Static Semantics Cheat Sheet

Kenton Lam

Friday July 3, 2020

Abstract

A quick reference on static semantics of the PL0 programming language. This is meant to be read alongside `PL0-SSemantics.pdf`. Written by Kenton Lam.

Abstract Syntax

$$
\begin{array}{rcl}
\mathsf{program} & ::= & \mathsf{block} \\
\mathsf{block} & ::= & \mathsf{blk(ds,s)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{ds} & ::= & \mathsf{id} \nrightarrow \mathsf{d} \\
\mathsf{d} & ::= & \mathsf{const(c)} \\
 & \mid & \mathsf{type(t)} \\
 & \mid & \mathsf{var(t)} \\
 & \mid & \mathsf{proc(block)} \\
\mathsf{c} & ::= & \mathsf{n} \mid \mathsf{id} \mid \mathsf{op}(-\_,\mathsf{c}) \\
\mathsf{t} & ::= & \mathsf{id} \mid [\mathsf{c} \,..\, \mathsf{c}]
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{s} & ::= & \mathsf{assign(lv,e)} \\
 & \mid & \mathsf{write(e)} \\
 & \mid & \mathsf{read(lv)} \\
 & \mid & \mathsf{call(id)} \\
 & \mid & \mathsf{if(e,s,s)} \\
 & \mid & \mathsf{while(e,s)} \\
 & \mid & \mathsf{list}(\mathit{seq}\ \mathsf{s})
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{lv} & ::= & \mathsf{id} \\
\mathsf{e} & ::= & \mathsf{n} \mid \mathsf{lv} \mid \mathsf{op(unary,e)} \mid \mathsf{op(binary,(e,e))}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{unary} & ::= & -\_ \\
\mathsf{binary} & ::= & \_+\_ \mid \_-\_ \mid \_*\_ \mid \_/\_ \mid \_=\_ \mid \_\neq\_ \mid \\
 & & \_<\_ \mid \_\leq\_ \mid \_>\_ \mid \_\geq\_
\end{array}
$$

Notation

- Abstract syntax is written in sans-serif font. For example the expression `y+1` in code is written as $\mathsf{y}+1$ here. This corresponds to expression nodes and statement nodes.
- Semantic constructs are written in *italics*. For example, the type $\mathit{ref}(T)$. These are most commonly types or other static checker constructs like the symbol table.

- The use of arrows is very precise:
  - $\rightarrow$ denotes functions,
  - $\mapsto$ denotes mapping values,
  - $\xrightarrow{e}$ denotes evaluates to, and
  - $\rightarrow$ with a vertical line in the centre denotes a mapping type (we will use $\twoheadrightarrow$ here because I can't type it).
- A bullet $\bullet$ is used to separate quantifiers from their predicate.
- $=$ denotes equivalence in the mathematical sense (a strong statement).

With the above in mind, we define a few more high-level pieces.

- A mapping M of $a \twoheadrightarrow b$ has some operations defined:
  - $\operatorname{dom}(M)$ returns the set of a keys in the mapping,
  - $M(a)$ returns the b value mapped to by the key a, and
  - $M_1 \oplus M_2$ returns a mapping containing the entries of both M_1 and M_2, with entries in M_2 overriding M_1 if both are present.
- *syms* is defined as a mapping $\mathsf{id} \twoheadrightarrow \mathit{SymEntry}$ of AST identifiers to symbol table entries, so $\operatorname{dom}(\mathit{syms})$ is the set of defined identifiers. A symbol table entry is a sum type defined as

$$\begin{aligned}\mathit{SymEntry} ::= {} & \mathit{ConstEntry}(T, \mathbb{Z}) \mid \mathit{TypeEntry}(T) \\ & \mid \mathit{VarEntry}(T) \mid \mathit{ProcEntry}(\mathsf{block}).\end{aligned}$$

- $\mathit{syms} \vdash \mathsf{e} : T$ means in the context of the symbol table *syms*, the expression e is well-typed and has the type T.

Declarations

There are four forms of declarations: constants, types, variables, and procedures. The notation $\mathit{syms} \vdash \mathit{WFDeclaration}(\mathsf{d})$ means that d is a well-formed declaration in the context of *syms*. Furthermore,

$$\mathit{entry}(\mathit{syms}, \mathsf{d}) = \mathit{ent}$$

is used to assign *ent* to the symbol table entry of d. This formally assigns a SymEntry to a particular declaration form d. Perhaps more rigorously, the statement

$$\mathit{entry}(\mathit{syms}, \mathsf{var}(\mathsf{t})) = \mathit{VarEntry}(\mathit{ref}(T))$$

means that a declaration of the form $\mathsf{var}(\mathsf{t})$ should have a corresponding VarEntry table when interpreted in the context of *syms*.

The declaration rules use pattern matching in their consequents, which means only expressions of a certain form can be well-formed declarations. This prevents us from declaring a type of, say, $1 + 10$.

Types

These rules concern the definition of types in PL0 program (e.g. type aliases and subrange types).

We introduce a function *typeof* such that $\mathit{typeof}(\mathsf{e}) = T$ means the given expression **is** (i.e. defines) the type T. This is at a higher level of abstraction than $\mathsf{e} : T$ which means e is a value of type T.

Blocks

These are perhaps the most complex because they must consider everything discussed already, as well as locally declared types, variables, and scope.

Note that a well-formed block cannot define an identifier more than once. This is represented (theoretically) by ds being a mapping. A block defines a new scope in which its local declarations shadow its parents identifiers if they have the same name. In doing so, it constructs symbol table entries from its declaration list.

The function *uses* takes a declaration, type or constant expression and returns the identifiers used by its types. For example, $uses(\mathsf{id}) = \{\mathsf{id}\}$ and $uses(\mathsf{var}(\mathsf{t})) = uses(\mathsf{t})$.

The $entryDecl(syms, \mathsf{ds}, \mathsf{d})$ function *defines* a symbol table entry for the declaration d in a context of *syms* augmented with only the declarations from ds which are used in d. The $uses(\mathsf{d})$ prevents mutual recursion in rule 6.2 between declarations in the same declaration list. Basically, this constructs a new context and offloads the work to *entry*.

The earlier *entry* function returns the appropriate symbol table entry for a given declaration in a given context. Importantly, this means that types and their SymEntries are constructed in the context they're defined. Putting this together, we get the following rules for well-formed blocks.

Rule 6.1 Well formed block

$$\frac{\begin{array}{c} ds\_uses = \{\mathsf{id}_1 \in \mathrm{dom}(\mathsf{ds}); \mathsf{id}_2 \in uses(\mathsf{ds}(\mathsf{id}_1)) \bullet \mathsf{id}_1 \mapsto \mathsf{id}_2\} \\ \neg\exists \mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet ((\mathsf{id} \mapsto \mathsf{id}) \in ds\_uses^+) \\ syms' = syms \oplus \{\mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet \mathsf{id} \mapsto entryDecl(syms, \mathsf{ds}, \mathsf{ds}(\mathsf{id}))\} \\ \forall \mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet (syms' \vdash WFDeclaration(\mathsf{ds}(\mathsf{id}))) \\ syms' \vdash WFStatement(\mathsf{s}) \end{array}}{syms \vdash WFBlock(\mathsf{blk}(\mathsf{ds},\mathsf{s}))}$$

Rule 6.2 EntryDecl

$$\frac{syms' = syms \oplus \{\mathsf{id} \in (\mathrm{dom}(\mathsf{ds}) \cap uses(\mathsf{d})) \bullet (\mathsf{id} \mapsto entryDecl(syms, \mathsf{ds}, \mathsf{ds}(\mathsf{id})))\}}{entryDecl(syms, \mathsf{ds}, \mathsf{d}) = entry(syms', \mathsf{d})}$$

Intuitively, the predicate of rule 6.1 does the following:

- Constructs a mapping of identifiers to the identifiers they use and computes its transitive closure (denoted by superscript $+$).
- Ensures that no identifier uses itself directly or indirectly, preventing recursion in the types.
- Constructs a new scope $syms'$ by adding the new declarations and computing their symbol table entries using *entryDecl* (discussed above).
- Ensures that every new declaration is well-formed in the new context.
- Ensures that in the new context, the statement list is well-formed.

If all of the above hold, the block as a whole is well-formed.

A *program* is well-formed if its block is well-formed in the context of the *predefined* context.

Rules

Types of Expressions

Rule 3.1 Integer value

$$\textit{syms} \vdash \mathsf{n} : \textit{int}$$

Rule 3.2 Symbolic constant

$$\frac{\mathsf{id} \in \mathrm{dom}(\textit{syms}) \quad \textit{syms}(\mathsf{id}) = \textit{ConstEntry}(T, v)}{\textit{syms} \vdash \mathsf{id} : T}$$

Rule 3.3 Variable identifier

$$\frac{\mathsf{id} \in \mathrm{dom}(\textit{syms}) \quad \textit{syms}(\mathsf{id}) = \textit{VarEntry}(T)}{\textit{syms} \vdash \mathsf{id} : T}$$

Rule 3.4 Unary negation

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{int}}{\textit{syms} \vdash \mathsf{op}(-\_,\mathsf{e}) : \textit{int}}$$

Rule 3.5 Binary operator

$$\frac{\textit{syms} \vdash \mathsf{e1} : T1 \quad \textit{syms} \vdash \mathsf{e2} : T2 \quad \textit{syms} \vdash \_ \odot \_ : T1 \times T2 \rightarrow T3}{\textit{syms} \vdash \mathsf{op}(\_ \odot \_,(\mathsf{e1},\mathsf{e2})) : T3}$$

Rule 3.6 Dereference

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{ref}(T)}{\textit{syms} \vdash \mathsf{e} : T}$$

Rule 3.7 Widen subrange

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{subrange}(T, i, j)}{\textit{syms} \vdash \mathsf{e} : T}$$

Rule 3.8 Narrow subrange

$$\frac{\textit{syms} \vdash \mathsf{e} : T \quad i \leq j \quad T \in \{\textit{int}, \textit{boolean}\}}{\textit{syms} \vdash \mathsf{e} : \textit{subrange}(T, i, j)}$$

Well-Formed Statements

Rule 4.1 Assignment

$$\frac{\textit{syms} \vdash \mathsf{lv} : \textit{ref}(T) \quad \textit{syms} \vdash \mathsf{e} : T}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{assign}(\mathsf{lv},\mathsf{e}))}$$

Rule 4.2 Procedure call

$$\frac{\mathsf{id} \in \mathrm{dom}(\textit{syms}) \quad \textit{syms}(\mathsf{id}) = \textit{ProcEntry}(\mathsf{block})}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{call}(\mathsf{id}))}$$

Rule 4.3 Read

$$\frac{\textit{syms} \vdash \mathsf{lv} : \textit{ref}(T) \quad (T = \textit{int} \vee T = \textit{subrange}(\textit{int}, i, j))}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{read}(\mathsf{lv}))}$$

Rule 4.4 Write

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{int}}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{write}(\mathsf{e}))}$$

Rule 4.5 Conditional

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{boolean} \quad \textit{syms} \vdash \textit{WFStatement}(\mathsf{s1}) \quad \textit{syms} \vdash \textit{WFStatement}(\mathsf{s2})}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{if}(\mathsf{e},\mathsf{s1},\mathsf{s2}))}$$

Rule 4.6 Iteration

$$\frac{\textit{syms} \vdash \mathsf{e} : \textit{boolean} \quad \textit{syms} \vdash \textit{WFStatement}(\mathsf{s})}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{while}(\mathsf{e},\mathsf{s}))}$$

Rule 4.7 Statement list

$$\frac{\forall \mathsf{s} \in \textit{elems}(\mathsf{ls}) \bullet (\textit{syms} \vdash \textit{WFStatement}(\mathsf{s}))}{\textit{syms} \vdash \textit{WFStatement}(\mathsf{list}(\mathsf{ls}))}$$

Well-Formed Declarations

Rule 5.1 Constant declaration

$$\frac{\textit{syms} \vdash \mathsf{c} \xrightarrow{e} v \quad \textit{syms} \vdash \mathsf{c} : T \quad T \in \{\textit{int}, \textit{boolean}\}}{\textit{syms} \vdash \textit{WFDeclaration}(\mathsf{const}(\mathsf{c}))}$$

Rule 5.2 Constant entry

$$\frac{\textit{syms} \vdash \mathsf{c} \xrightarrow{e} v \quad \textit{syms} \vdash \mathsf{c} : T \quad T \in \{\textit{int}, \textit{boolean}\}}{\textit{entry}(\textit{syms}, \mathsf{const}(\mathsf{c})) = \textit{ConstEntry}(T, v)}$$

Rule 5.3 Type declaration

$$\frac{\textit{syms} \vdash \textit{typeof}(\mathsf{t}) = T}{\textit{syms} \vdash \textit{WFDeclaration}(\mathsf{type}(\mathsf{t}))}$$

Rule 5.4 Type entry

$$\frac{\textit{syms} \vdash \textit{typeof}(\mathsf{t}) = T}{\textit{entry}(\textit{syms}, \mathsf{type}(\mathsf{t})) = \textit{TypeEntry}(T)}$$

Rule 5.5 Variable declaration

$$\frac{\textit{syms} \vdash \textit{typeof}(\mathsf{t}) = T}{\textit{syms} \vdash \textit{WFDeclaration}(\mathsf{var}(\mathsf{t}))}$$

Rule 5.6 Variable entry

$$\frac{\textit{syms} \vdash \textit{typeof}(\mathsf{t}) = T}{\textit{entry}(\textit{syms}, \mathsf{var}(\mathsf{t})) = \textit{VarEntry}(\textit{ref}(T))}$$

Rule 5.7 Procedure declaration

$$\frac{\textit{syms} \vdash \textit{WFBlock}(\mathsf{block})}{\textit{syms} \vdash \textit{WFDeclaration}(\mathsf{proc}(\mathsf{block}))}$$

Rule 5.8 Procedure entry

$$\frac{\textit{syms} \vdash \textit{WFBlock}(\mathsf{block})}{\textit{entry}(\textit{syms}, \mathsf{proc}(\mathsf{block})) = \textit{ProcEntry}(\mathsf{block})}$$

Constant Evaluation Rules

Rule 5.9 Integer constant

$$\frac{0 \leq \mathsf{n} \leq maxint}{syms \vdash \mathsf{n} \xrightarrow{e} \mathsf{n}}$$

Rule 5.10 Constant identifier

$$\frac{\begin{array}{c}\mathsf{id} \in \mathrm{dom}(syms) \\ syms(\mathsf{id}) = ConstEntry(T, v)\end{array}}{syms \vdash \mathsf{id} \xrightarrow{e} v}$$

Rule 5.11 Negated constant

$$\frac{\begin{array}{c}syms \vdash \mathsf{c} : int \\ syms \vdash \mathsf{c} \xrightarrow{e} v\end{array}}{syms \vdash \mathsf{op}(-\_,\mathsf{c}) \xrightarrow{e} -v}$$

Well-Formed Types

Rule 5.12 Type identifier

$$\frac{\begin{array}{c}\mathsf{id} \in \mathrm{dom}(syms) \\ syms(\mathsf{id}) = TypeEntry(T)\end{array}}{syms \vdash typeof(\mathsf{id}) = T}$$

Rule 5.13 Subrange type

$$\frac{\begin{array}{ccc}syms \vdash \mathsf{c0} : T & syms \vdash \mathsf{c0} \xrightarrow{e} v0 & v0 \leq v1 \\ syms \vdash \mathsf{c1} : T & syms \vdash \mathsf{c1} \xrightarrow{e} v1 & T \in \{int, boolean\}\end{array}}{syms \vdash typeof([\mathsf{c0}\,..\,\mathsf{c1}]) = subrange(T, v0, v1)}$$

Well-Formed Blocks

Rule 6.1 Well formed block

$$\frac{\begin{array}{c}ds\_uses = \{\mathsf{id}_1 \in \mathrm{dom}(\mathsf{ds}); \mathsf{id}_2 \in uses(\mathsf{ds}(\mathsf{id}_1)) \bullet \mathsf{id}_1 \mapsto \mathsf{id}_2\} \\ \neg\exists \mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet ((\mathsf{id} \mapsto \mathsf{id}) \in ds\_uses^{+}) \\ syms' = syms \oplus \{\mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet \mathsf{id} \mapsto entryDecl(syms, \mathsf{ds}, \mathsf{ds}(\mathsf{id}))\} \\ \forall \mathsf{id} \in \mathrm{dom}(\mathsf{ds}) \bullet (syms' \vdash WFDeclaration(\mathsf{ds}(\mathsf{id}))) \\ syms' \vdash WFStatement(\mathsf{s})\end{array}}{syms \vdash WFBlock(\mathsf{blk}(\mathsf{ds},\mathsf{s}))}$$

Rule 6.2 EntryDecl

$$\frac{syms' = syms \oplus \{\mathsf{id} \in (\mathrm{dom}(\mathsf{ds}) \cap uses(\mathsf{d})) \bullet (\mathsf{id} \mapsto entryDecl(syms, \mathsf{ds}, \mathsf{ds}(\mathsf{id})))\}}{entryDecl(syms, \mathsf{ds}, \mathsf{d}) = entry(syms', \mathsf{d})}$$

Well-Formed Main Program

Rule 7.1 Well-formed main program

$$\frac{predefined \vdash WFBlock(\mathsf{block})}{WFProgram(\mathsf{block})}$$

The symbol table for the predefined identifiers[10] is

$$\begin{aligned}predefined = \{&\mathsf{int} \mapsto TypeEntry(int), \\ &\mathsf{boolean} \mapsto TypeEntry(boolean), \\ &\mathsf{false} \mapsto ConstEntry(boolean, 0), \\ &\mathsf{true} \mapsto ConstEntry(boolean, 1)\}\ .\end{aligned}$$