Pen-and-paper semantics for P4

Anoud Alshnakat
Didrik Lundberg

April 28, 2021

This is a pen-and-paper semantics of P4, based on the official P4 specification and inspired by Core P4 [**doenges2021petr4**].

1 Syntax

1.1 Types

x	string
num	natural number
$index$	natural number
b	boolean
n	integer
i, j, k, l	indices

Figure 1: Primitive Types

The types shown in Figure 1 are the subset of P4 types included in this formalization and their standard designations, plus the numerals num and the indices which are not P4 types, but used throughout this formalization. A string x can be used as a function name or a variable name. The integer n is a 64-bit word.

1.2 Expressions

Our formalization includes a subset of the full set of P4 expressions found in Section 8 of the P4 specification.

exp	$::=$		expression
	$\mid$	b	boolean value
	$\mid$	n	integer value
	$\mid$	x	variable/function name
	$\mid$	$\ominus exp$	unary operation
	$\mid$	$exp_1 \oplus exp_2$	binary operation
	$\mid$	$\textbf{call}\, x(exp_1, \,..\,, exp_i)$	function call
	$\mid$	$\textbf{exec}\, stmt$	function execution
	$\mid$	(exp) S	

Figure 2: P4 Expressions

The expressions included are shown in Figure 2. First, an expression can be a Boolean or an integer (collectively referred to as constants), or a string. There exist unary and binary arithmetic operations, where the semantics of the individual operations are defined on some subset of the constants. The function call is built from the function name x, and a list of arguments (expressions). Function execution is not found in any P4 program, but is rather an artifact of our semantics, signifying the in-progress execution of the body of a called function. The concrete syntax of unary and binary operations is found in Appendix A.

1.3 Statements

See Section 11 of the P4 specification.

stmt	$::=$		statement
	$\mid$	$\emptyset_{\mathrm{stmt}}$	empty statement
	$\mid$	$x := \mathit{exp}$	assignment
	$\mid$	$\textbf{if}\ \mathit{exp}\ \textbf{then}\ \mathit{stmt}_1\ \textbf{else}\ \mathit{stmt}_2$	conditional
	$\mid$	$\mathit{decl}\ x$	declaration
	$\mid$	$\{\mathit{stmt}\}$	block
	$\mid$	$[\mathit{stmt}]$	block in progress
	$\mid$	$\textbf{return}\ \mathit{exp}$	return
	$\mid$	$\mathit{stmt}_1; \mathit{stmt}_2$	sequence

Figure 3: P4 Statements

1.4 Execution State

A scope is a partial function that represents a mapping from variable names as strings to values in a certain scope. The operations that be done on the scopes are:

1. Finding the domain of a scope. (add math representation here)
2. and updating some variable (or multiple variables) mapping.

The list of scopes ε is a list where the global scope variables are stored at the bottom in location $\varepsilon[0]$, whereas the variables of the current scope that being executed are stored on the top ε, and it is created whenever a new scope{} is entered. The operations that can be done on the list of scopes are (not limited to):

1. Adding a new scope to the list. (add math representation here)
2. Concatenating two scope frames together.
3. Updating a scope in a certain location (index).

The call stack E is a list of stack frames, used wherever a function call occurs. Whenever the execution reaches a function call, the caller scope in location ε[i] will be stored in the E, and later retrieved from E and store back in ε[i]. The operations that can be done on the list of scopes are (not limited to):

1. Adding a list of scopes to the call stack.

The state memory σ consists of a tuple of (ε, E), to represent the live variable values of the program at a certain execution point.

The status **Running** represents that the program is executing under regular circumstances. **TypeError** represents a crash caused by some badly-typed part of the program. **Return** is used when the **return** statement returns a constant inside a function.

The execution state S holds a the state memory σ and the status. Note that none of these constructs are laid out in the P4 specification, but rather made up by yours truly in order to obtain a formal P4 semantics.

$$
\begin{array}{lcll}
\mathit{status} & ::= & & \text{execution status} \\
 & | & \textbf{Running} & \\
 & | & \textbf{Return}\ \mathit{exp} & \\
 & | & \textbf{TypeError} & \\
 & & & \\
S & ::= & & \text{execution state} \\
 & | & (\sigma, \mathit{status}) &
\end{array}
$$

Figure 4: P4 Execution State

2 Semantics

2.1 Expressions

$$\boxed{[exp](\sigma) \rightsquigarrow [exp'](\sigma')} \quad \text{expression semantics}$$

$$\frac{\begin{array}{l} index = min\{j.x \in \mathrm{dom}(\varepsilon[j])\} \\ scope = \varepsilon[index] \\ V = scope(x) \\ \mathrm{is\_const}(V) \end{array}}{[x]\sigma \rightsquigarrow [V]\sigma} \quad \textsc{exp\_lookup}$$

$$\begin{array}{l} exp'_1, \,..\,, exp'_j + +exp''_1, \,..\,, exp''_k = exp_1, \,..\,, exp_i \\ \mathrm{is\_consts}(exp_1, .., exp_i) \\ exp = \mathbf{hd}\, exp''_1, \,..\,, exp''_k \\ [exp]\sigma \rightsquigarrow [exp']\sigma' \\ num = \mathbf{length}\,(exp'_1, \,..\,, exp'_j) \end{array}$$

«no parses (char 72): exp'''1 , .. , exp'''l =' update (exp' , num , (exp1 , .. , expi))*** »

$$\overline{[\mathbf{call}\, x(exp_1, \,..\,, exp_i)]\sigma \rightsquigarrow [\mathbf{call}\, x(exp'''_1, \,..\,, exp'''_l)]\sigma'}$$

$$\frac{\begin{array}{l} \mathrm{is\_consts}(exp_1, .., exp_i) \\ (stmt, x_1, \,..\,, x_i) = F(x) \\ scope' = \forall i.(x_i \mapsto exp_i)\emptyset_{\mathrm{scope}} \\ G\_scope = \varepsilon[0] \\ \varepsilon' = scope' :: [G\_scope] \\ E' = \mathrm{tl}(\varepsilon) :: E \end{array}}{[\mathbf{call}\, x(exp_1, \,..\,, exp_i)](\varepsilon, E) \rightsquigarrow [\mathbf{exec}\, stmt](\varepsilon', E')} \quad \textsc{exp\_func\_call\_args}2$$

$$\frac{[stmt](\sigma, \mathbf{Running}) \to [stmt'](\sigma', \mathbf{Running})}{[\mathbf{exec}\, stmt]\sigma \rightsquigarrow [\mathbf{exec}\, stmt']\sigma'} \quad \textsc{exp\_func\_exec}$$

$$\frac{\mathrm{is\_const}(exp)}{[\mathbf{exec}\, \emptyset_{\mathrm{stmt}}]\sigma \rightsquigarrow [exp]\sigma'} \quad \textsc{exp\_func\_ret\_exp}$$

$$\frac{[exp]\sigma \rightsquigarrow [exp']\sigma'}{[\ominus exp]\sigma \rightsquigarrow [\ominus exp']\sigma'} \quad \textsc{exp\_unop}$$

$$\frac{[exp]\sigma \rightsquigarrow [exp'']\sigma'}{[exp \oplus exp']\sigma \rightsquigarrow [exp'' \oplus exp']\sigma'} \quad \textsc{exp\_binop}1$$

$$\frac{\begin{array}{l} \mathrm{is\_const}(exp) \\ [exp']\sigma \rightsquigarrow [exp'']\sigma' \end{array}}{[exp \oplus exp']\sigma \rightsquigarrow [exp \oplus exp'']\sigma'} \quad \textsc{exp\_binop}2$$

Figure 5: P4 Expression Evaluation Semantics

The small-step semantics for reducing expressions is shown in Figure 5.

In the EXP_LOOKUP rule, the first antecedent is a function $index = min\{j.x \in \mathrm{dom}(\varepsilon[j])$ ensures that the variable name x is evaluated in the uppermost (i.e. most recent entered) scope of ε where it is declared, by preventing x to be in the domain of any *scope* higher in ε than the one used for variable resolution. This agrees with the description in Sections 6.8 and 10.2 of the P4 specification. The value of this variable is then resolved, and checked to be a constant.

The EXP_FUNC_CALL_ARGS1 rule reduces the leftmost function argument which has yet to be reduced to a constant with one expression evaluation step. The first two antecedents divide the list of arguments into two sub-lists, where the prefix must contain all constants. The head of the suffix is then reduced with one expression small-step, after which the corresponding index in the original list of arguments is update with the resulting expression.

The EXP_FUNC_CALL_ARGS2 rule is used when all of the function arguments have been reduced to constants using EXP_FUNC_CALL_ARGS1 (or if they were all constants to begin with). The constants are assigned to their respective argument names in a fresh scope, after which this scope is put on top of the global scope $\varepsilon[0]$ in order to form the new current list of scope frame ε'. The old current list of scopes frames ε is then saved on top of the call stack E to be used later when returning from the function call, and the function call statement is reduced to **exec** *stmt* - in-progress execution of the function body *stmt* (obtained from the function map F, which holds mappings between function names x and tuples of function bodies and lists of their argument names). Note that this rule also covers the case of a function call with no arguments.

The EXP_FUNC_EXEC rule reduces the function body of in-progess execution with one small-step statement reduction.

The EXP_FUNC_RET_EXP rule reduces finished (empty) in-progess execution with status **Return** *exp* to *exp*, provided *exp* is a constant. This also changes the status to **Running**.

Section 8.1 of the P4 specification states that expressions are evaluated left-to-right. Accordingly, the rules for binary operations - EXP_BINOP1 and EXP_BINOP2 - are split up so that (small-step) reduction of the second operand requires that the first operand has been completely reduced to a constant. This is trivial for unary operations (EXP_UNOP).

2.2 Statement Execution

The SOS of the statements is shown in Figure 6

The STMT_RET_EXP rule implements one reduction to the expression at a time, to simplify the expression until it reduces to a constant. Once the **return** statement appears to return a constant the rule STMT_RET_CONST contains the antecedents that are required for such operation. The global scope is always stored in the location zero of the list of scopes, i.e. $\varepsilon[0]$. It is fetched and concatenated with most recent caller that was stored on top of the call stack E. Thus, this concatenation will generate a ε that has the same shape to the one before the function being called (of course before reaching the **return** statement some global variables could have been changed during function evaluation, that is the reason why we say that the shape is the same but not the variable mappings in the global scope). The status will be changed to **Return V** where it will be handled later in an other rule.

The STMT_ASS_EXP rule implements one reduction to the expression at a time, to simplify the expression until it reduces to a constant. The STMT_ASS_CONST rule handles assignment statement. In general, the variables that the program can assign values to it should be in the global scope or the current scope of the frame, thus we need to look up into the current list of scopes ε, but never into E. So the antecedent $index = max\{j.x \in \mathrm{dom}(\varepsilon[j])$ fetches the proper index that locates the variable location, it should be the one in the lowermost part of the the current list of scopes ε (i.e. oldest entry). In the last antecedent, $(x \longrightarrow V)scope$ updates the mapping of the variable name x to the new value i.e. (constant V) in the proper frame location, that indeed lies in the current list of scope frame. One step reduction in this rule results an updated current list of scopes, and an empty statement to execute.

The STMT_SEQ1 and STMT_SEQ2 rules are trivial to understand, they are pretty standard.

The STMT_SEQ3 handles the **return** statement when it does not occur at the end of the the function code. The next statement to reduce will be empty and the status will be changed to **Return** that will be handled later.

The STMT_COND1, STMT_COND2 and STMT_COND3 rules are pretty trivial to understand.

The STMT_DECL rule is a transition reduction for the declaration statements. Whenever a variable is declared, it will updated in the most recent scope and reduce the transition to an empty statement. The most recent/newest scope is fetched simply by checking the length of the list of scopes ε.

The **{}** brackets indicates a block, while the **[ ]** brackets indicates a block being executed. Whenever entering a block, in rule STMT_BLOCK_ENTER, a new empty scope is added to the list of scopes, and then the **{}** brackets are switched the executing ones **[ ]** to pinpoint the fact that the statements can be executed now.

STMT_BLOCK_EXEC rule shows a small step reduction for the statements inside a block.

STMT_BLOCK_EXIT rule handles the block termination. Whenever a block contains an empty statement, then it is necessary to also remove the corresponding scope from the list of scopes ε which in the case of blocks is the most recent one.

$\boxed{[stmt]S \to [stmt']S'}$ statement semantics

$$\frac{[exp]\sigma \rightsquigarrow [exp']\sigma'}{[\mathbf{return}\ exp](\sigma, \mathbf{Running}) \to [\mathbf{return}\ exp'](\sigma', \mathbf{Running})} \quad \textsc{stmt\_ret\_exp}$$

$$\frac{\begin{array}{l}\text{is\_const}(V)\\ G\_scope = \varepsilon[0]\\ \varepsilon' :: E' = E\\ \varepsilon'' = (\varepsilon') + +([G\_scope])\end{array}}{[\mathbf{return}\, V]((\varepsilon, E), \mathbf{Running}) \to [\emptyset_{\text{stmt}}]((\varepsilon'', E'), \mathbf{Return}\, V)} \quad \textsc{stmt\_ret\_const}$$

$$\frac{\begin{array}{l}\text{is\_const}(V)\\ index = max\{j.x \in \text{dom}(\varepsilon[j])\}\\ scope = \varepsilon[index]\\ \varepsilon' = (index \mapsto (x \mapsto V)scope)\varepsilon\end{array}}{[x := V]((\varepsilon, E), \mathbf{Running}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{Running})} \quad \textsc{stmt\_ass\_const}$$

$$\frac{[exp]\sigma \rightsquigarrow [exp']\sigma'}{[x := exp](\sigma, \mathbf{Running}) \to [x := exp'](\sigma', \mathbf{Running})} \quad \textsc{stmt\_ass\_exp}$$

$$\frac{[stmt_1](\sigma, \mathbf{Running}) \to [stmt_1'](\sigma', \mathbf{Running})}{[stmt_1; stmt_2](\sigma, \mathbf{Running}) \to [stmt_1'; stmt_2](\sigma', \mathbf{Running})} \quad \textsc{stmt\_seq}1$$

$$\frac{}{[\emptyset_{\text{stmt}}; stmt](\sigma, \mathbf{Running}) \to [stmt](\sigma, \mathbf{Running})} \quad \textsc{stmt\_seq}2$$

$$\frac{[stmt_1](\sigma, \mathbf{Running}) \to [stmt_1'](\sigma', \mathbf{Return}\ exp)}{[stmt_1; stmt_2](\sigma, \mathbf{Running}) \to [\emptyset_{\text{stmt}}](\sigma', \mathbf{Return}\ exp)} \quad \textsc{stmt\_seq}3$$

$$\frac{[exp]\sigma \rightsquigarrow [exp']\sigma'}{[\mathbf{if}\ exp\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{Running}) \to [\mathbf{if}\ exp'\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma', \mathbf{Running})} \quad \textsc{stmt\_cond}1$$

$$\frac{b = \text{True}}{[\mathbf{if}\ b\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{Running}) \to [stmt_1](\sigma, \mathbf{Running})} \quad \textsc{stmt\_cond}2$$

$$\frac{b = \text{False}}{[\mathbf{if}\ b\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{Running}) \to [stmt_2](\sigma, \mathbf{Running})} \quad \textsc{stmt\_cond}3$$

$$\frac{\begin{array}{l}index = length(\varepsilon)\\ scope = \varepsilon[index]\\ \varepsilon' = (index \mapsto (x \mapsto 0)scope)\varepsilon\end{array}}{[decl\ x]((\varepsilon, E), \mathbf{Running}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{Running})} \quad \textsc{stmt\_decl}$$

$$\frac{\varepsilon' = \emptyset_{\text{scope}} :: \varepsilon}{[\{stmt\}]((\varepsilon, E), \mathbf{Running}) \to [[stmt]]((\varepsilon', E), \mathbf{Running})} \quad \textsc{stmt\_block\_enter}$$

$$\frac{[stmt](\sigma, \mathbf{Running}) \to [stmt'](\sigma', \mathbf{Running})}{[[stmt]](\sigma, \mathbf{Running}) \to [[stmt']](\sigma', \mathbf{Running})} \quad \textsc{stmt\_block\_exec}$$

$$\frac{\varepsilon' = \text{tl}(\varepsilon)}{[[\emptyset_{\text{stmt}}]]((\varepsilon, E), \mathbf{Running}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{Running})} \quad \textsc{stmt\_block\_exit}$$

Figure 6: P4 Statement Execution Semantics

A Concrete Syntax of Operations

$$
\begin{array}{lcll}
\ominus & ::= & & \\
 & | & ! & \text{boolean negation} \\
 & | & \neg & \text{bitwise complement} \\
 & | & - & \text{signed negation} \\
 & | & + & \text{unary plus}
\end{array}
$$

Figure 7: P4 Unary Operations

The unary expressions included are shown in Figure 7. These include all of the unary operations in P4. Boolean negation is only defined on Booleans, the other operations have their standard meanings (note that unary plus is a no-op).

$$
\begin{array}{lcll}
\oplus & ::= & & \\
 & | & \times & \text{multiplication} \\
 & | & / & \text{division} \\
 & | & \mathrm{mod} & \text{modulo} \\
 & | & + & \text{addition} \\
 & | & - & \text{subtraction} \\
 & | & \ll & \text{left-shift} \\
 & | & \gg & \text{right-shift} \\
 & | & \leq & \text{less or equal} \\
 & | & \geq & \text{greater or equal} \\
 & | & < & \text{less} \\
 & | & > & \text{greater} \\
 & | & \neq & \text{not equal} \\
 & | & = & \text{equal} \\
 & | & \& & \text{bitwise and} \\
 & | & \veebar & \text{bitwise xor} \\
 & | & | & \text{bitwise or} \\
 & | & \wedge & \text{binary and} \\
 & | & \vee & \text{binary or}
\end{array}
$$

Figure 8: P4 Binary Operations

The binary expressions included are shown in Figure 7. These include all of the binary operations in P4.