The p4ott P4 Formalization

Anoud Alshnakat
Didrik Lundberg

October 6, 2021

This is a description of the p4ott formalization of P4, which includes a syntax and a strictly small-step style semantics. It is based on the official P4 specification and inspired by Core P4 [**doenges2021petr4**].

p4ott is constructed using the ott tool. ott files can then be exported to LaTeX commands (used in this document) as well as to the HOL4, Isabelle/HOL and Coq interactive theorem provers (of which only the first is currently supported).

1 Syntax

1.1 Types

x, f, msg, a, $table\_name$	string
b	boolean
n_w	integer
d	direction
i	natural number
m, n, o	indices

Figure 1: Variables

The variables shown in Figure 1 are standard designations for variables of P4 base types included in p4ott, plus the numerals i and the indices m, n, o which are not part of the P4 syntax, but used on a meta-level throughout this formalization. Depending on the context, strings are denoted with x (variable name), f (function or field name) or msg (error message). The integer n_w is a 64-bit word.

Types are sometimes explicitly referenced in the syntax, e.g. in declaration statements. The notation for this is shown in Figure 2. Subscript t is used to clarify the notation refers to a type, as opposed to a variable of that type. Declared instances of composite types are stored in the type environment T.

bt	::=		base types
	|	bool_t	
	|	int_t	
t	::=		types
	|	bt	
	|	$\text{struct}_\text{t}\ t_1, \ldots, t_n$	

Figure 2: Types

1.2 Expressions

`p4ott` includes a subset of the full set of P4 expressions found in Section 8 of the P4 specification, shown in Figure 3.

e	::=		expression
	|	v	constant value
	|	x	variable name
	|	$\{e_1, .., e_n\}$	expression list
	|	$e.e'$	field access
	|	$\ominus e$	unary operation
	|	$e_1 \oplus e_2$	binary operation
	|	**call** $f(e_1, .., e_n)$	function call
	|	**exec** $stmt$	function execution
	|	(e)	

Figure 3: P4 Expressions

First, an expression can be a Boolean or an integer (collectively referred to as constant values v), or a string. Lists of expressions can be used in declarations of variables of struct types, whose fields may be accessed. There exist unary and binary arithmetic operations, where the semantics of the individual operations are defined on some subset of the constants[1]. The function call is built from the function name f, and a list of arguments (expressions). In-progress execution of the body of a called function, **exec** *stmt*, is not a part of the P4 syntax, but is rather an artifact of our small-step semantics.

[1]The concrete syntax of unary and binary operations is found in Appendix A

1.3 Statements

`p4ott` includes a subset of the full set of P4 statements found in Section 11 of the P4 specification, shown in Figure 4. They are mostly standard, apart from the following: the in-progress block is an artifact of our small-step semantics. The **verify** statement (here a statement and not an extern function as in Section 12.7 of the P4 specification) can be found uniquely in a parser block. It asserts the expressions e and if it holds, does nothing. If e does not hold, it jumps to a rejecting parser state with error message being the result of evaluating e'. The **transition** statement continues execution at a new parser state p.

stmt	::=		statement
	|	$\emptyset_{\text{stmt}}$	empty statement
	|	$lval := e$	assignment
	|	**if** e **then** $stmt_1$ **else** $stmt_2$	conditional
	|	**decl** $x\, t$	declaration
	|	$\{stmt\}$	block
	|	$[stmt]$	block in progress
	|	**return** e	return
	|	$stmt_1; stmt_2$	sequence
	|	**verify** $e\ e'$	verify
	|	**transition** p	transition
	|	**apply** $table\_name\ e$	apply

Figure 4: P4 Statements

The assignment can assign to *lval*s (shown in Figure 5), which include variables identified by their names, and struct fields, which are identified by the struct and field names, similar to the field access expression.

lval	::=		
	|	x	variable name
	|	**null**	null variable
	|	$lval.f$	field access
	|	$(lval)$	

Figure 5: P4 l-values

1.4 Execution State

The P4 execution state is shown in Figure 6. Note that nothing like this is described in the P4 specification, so it is entirely an artifice of the `p4ott` implementation. In short, the execution state s is a tuple of the state memory σ and the state status t. The state memory σ consists of a tuple (ε, E), where ε is a stack of scopes γ which hold the values of variables which are currently visible, and E holds variable mappings which belong to previous caller contexts.

t	::=		execution status
	|	**R**	
	|	**Ret** v	
	|	p	
	|	$\bot$	
s	::=		execution state
	|	(σ, t)	

Figure 6: P4 Execution State

More formally, a scope $\gamma : X \hookrightarrow V$ is a partial function from variable names $x \in X$ to constant values $v \in V$. The following operations can be performed on γ:

- $\text{dom}(\gamma)$: Gets the domain of γ: obtains the set of variable names $x \in X$ which are mapped to values in γ.
- $(x \mapsto v)\ \gamma$: Updates a variable mapping in γ: yields the scope γ', which is just γ where x instead maps to v. By writing $\forall i \leq n.\ (x_i \mapsto v_i)\ \gamma$ we extend this to lists of mappings from variable names to values.

A frame ε is a stack of scopes where the global scope γ_G is located at the bottom; that is, in location $\varepsilon[0]$. The current scope - that which was most recently entered by execution - is stored on the top of ε (note that this indexing is the reverse of what you would expect from a list). Whenever a new block (delineated by {}) is entered, a new fresh scope $\gamma_\emptyset$ is pushed onto the frame ε. The following operations can be performed on a frame ε:

- $\gamma :: \varepsilon$: Pushes a scope γ on top of ε.
- $(i \mapsto \gamma)\ \varepsilon$: Updates the scope located at index i of ε by setting it to γ.

The call stack E is a stack of frames used whenever a function call occurs. When a function call is executed, the frame ε (minus the global scope γ_G) of the caller will be pushed onto E. When the callee function finishes execution and returns, ε will be popped from E and pushed onto a frame containing only γ_G. Note that this means that the same γ_G is kept throughout function calls, and updates to it are passed along accordingly. The following operations can be performed on E:

- $\varepsilon :: E$: Pushes a frame ε onto the call stack E.

The status **R** represents that the program is executing under regular circumstances. **Ret** v is used when the **return** statement returns a constant v at the end of a function call. The status p signifies transition to a new parser state inside the parser - a named state in the case of **Trans** x, or

a final state (p_{fin}) in the case of **Accept** or **Reject** . $\perp$ represents a crash or undefined behaviour, for example caused by some badly-typed part of the program.

In addition to the above, there's also a function map F mapping function names to tuples of their bodies and argument names, a parser map P mapping parser state names to their bodies and a type environment T. These are assumed to be static, and are therefore not part of the execution state.

2 Semantics

2.1 Expressions

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

$$\frac{v = \text{lookup}_{\text{v}}(\varepsilon, x)}{[x]((\varepsilon, E), \mathbf{R}) \rightsquigarrow [v]((\varepsilon, E), \mathbf{R})} \quad \text{E\_LOOKUP}$$

$$\frac{\begin{array}{l}(stmt, (x_1, d_1), .., (x_n, d_n)) = F(f) \\ \forall d,\, e,\, i \leqslant n.\, d = [d_1, .., d_n][i] \wedge e = [e_1, .., e_n][i] \implies ((d \in \{\circ, \downarrow\} \implies \text{is\_const}\, e) \wedge (d \in \{\updownarrow, \uparrow\} \implies \text{is\_var}\, e)) \\ \forall d,\, e,\, x,\, i \leqslant n.\, x = [x_1, ..., x_n][i] \wedge e = [e_1, ..., e_n][i] \wedge d = [d_1, ..., d_n][i] \implies \gamma'(x) = \begin{cases} (\text{lookup}_{\text{v}}(\varepsilon, e), e) & \text{if } d \in \{\uparrow, \updownarrow\} \\ (e, \perp) & \text{if } d \in \{\downarrow, \circ\} \end{cases} \\ \gamma_G = \varepsilon[0] \\ \varepsilon' = \gamma' :: [\gamma_G] \\ \varepsilon'' = \text{tl}(\varepsilon) \\ E' = (\varepsilon'', f) :: E \end{array}}{[\textbf{call}\, f(e_1, .., e_n)]((\varepsilon, E), \mathbf{R}) \rightsquigarrow [\textbf{exec}\, stmt]((\varepsilon', E'), \mathbf{R})}$$

$$\frac{[stmt](\sigma, \mathbf{R}) \rightarrow [stmt'](\sigma', \mathbf{R})}{[\textbf{exec}\, stmt](\sigma, \mathbf{R}) \rightsquigarrow [\textbf{exec}\, stmt'](\sigma', \mathbf{R})} \quad \text{E\_FUNC\_EXEC}$$

$$\frac{}{[\textbf{exec}\, \emptyset_{\text{stmt}}](\sigma, \textbf{Ret}\ v) \rightsquigarrow [v](\sigma', \mathbf{R})} \quad \text{E\_FUNC\_RET}$$

$$\frac{v = \textbf{struct}\, \{x_1 = v_1; \, ...\, ; x_n = v_n\}(x)}{[\textbf{struct}\, \{x_1 = v_1; \, ...\, ; x_n = v_n\}.x](\sigma, \mathbf{R}) \rightsquigarrow [v](\sigma, \mathbf{R})} \quad \text{E\_ACC}$$

Figure 7: P4 Expression Evaluation Semantics

The semantics of expressions is shown in Figure 7[2][3].

In the E_LOOKUP rule, the first antecedent states that $i = \max\{j.\ x \in \text{dom}(\varepsilon[j])\}$, which ensures that the variable name x is evaluated in the uppermost (i.e. most recently entered) scope

[2]The semantics for reducing concrete arithmetic operations is standard and covers everything found in Appendix A
[3]Rules for reducing expressions in all contexts can be found in Appendix B

of ε where it can be found. 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 E_FUNC_CALL_NEWFRAME rule is used when all of the function arguments have been reduced to constants. 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 γ_G in order to form the new current frame ε'. The old current frame ε (minus γ_G) 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 f 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 E_FUNC_EXEC rule reduces the function body of in-progess execution with one statement reduction, and the E_FUNC_RET rule reduces finished (empty) in-progress execution with status **Return** e to e, provided e is a constant. This also changes the status to **Running**.

The E_ACC rule is used to access the values of fields in structs.

2.2 Calling convention

The calling conventions can be directioned or directionless. The direction can be either IN, OUT or INOUT. IN direction summary:

1. Should not be used on the left hand of assignment
2. Shouldn't be passed to a function without using the proper calling convention IN
3. Initialized by copying the value of the corresponding argument when the invocation is executed.

OUT summary:

1. usually uninitialized, and treated as l-values. after the execution of the call, the value of the OUT parameters copied to the corresponding location of the l-value. OUT parameters are initialized in the following cases
   (a) if the types are header or header_union, OUT parameter is set to "invalid"
   (b) if the type is a header stack, then all elements of the header stack set to "invalid" and the next index is initialized to 0.
   (c) if the type is compound (e.g. struct or tuple) apply the rules recursively to its members.
   (d) if any any other type than listed above (e.g. bit <W>), then it doesn't need any predictable value.

INOUT summary:

1. this type of parameters are both IN and OUT.
2. it must be an l-value, which means it can be assigned to a value.

NO direction summary:

1. those parameters are known at compile time.
2. it also can be an action parameter, can be set by the control plane.
3. it also can be an action parameter that set directly by an other action, then the behaviour will be like IN parameter.

The direction d can be $\downarrow$ denotes IN, $\uparrow$ denotes OUT, $\updownarrow$ denotes INOUT, $\circ$ denotes directionless. Thus, d ::= $\downarrow \mid \uparrow \mid \updownarrow \mid \circ$

In the function calls, it requires extending the scope type to be as following; $\gamma : X \hookrightarrow V * (X \cup \{\perp\})$

We shall also modify the stack E to be a list of tuples of 2 members. First member is the frame, while second member is the function name as in (ε, F_name). Define a a few new operations:

1. operation $\varepsilon[x \longmapsto v]$ to update a variable x with value v in the frame ε. This should be equivalent to the three premises in STMT_ASS_V
2. operation $lookup_t(\varepsilon, x)$ to return a tuple (y, d) that that variable x is mapped to in frame ε.
3. operation $lookup_v(\varepsilon, x)$ to return the value v of the tuple $(y, x \cup \perp)$, which is whatever variable x is mapped to in frame ε.

$$
\begin{array}{lll}
d & ::= & \text{the direction of the calling convention} \\
 & \mid & \downarrow \\
 & \mid & \uparrow \\
 & \mid & \updownarrow \\
 & \mid & \circ
\end{array}
$$

Figure 8: direction syntax

$$
\begin{array}{lll}
F_n & ::= & \text{called function name} \\
 & \mid & \bot \\
 & \mid & f
\end{array}
$$

Figure 9: called function name list

2.3 Statement Execution

The semantics of the statements is shown in Figures 11 and 10[4].

The STMT_DECL is used to reduce the **decl** statement, which has the effect of declaring variable mappings in the current (topmost) scope. The newly declared variable is given an uninitialized value, denoted by ?.

The STMT_ASS_V rule handles the assignment statement. In general, the variables that the program can assign values to are in the global scope γ_G or the current frame, thus we need to look up mappings in the current frame ε, but never in E. So the antecedent $i = \max\{j.\ x \in \mathrm{dom}(\varepsilon[j])$ obtains the index i of the uppermost (i.e. most recently entered) scope in the current frame ε containing the variable. In the last antecedent, $(x \longrightarrow v)\gamma$ updates the mapping of the variable name x to the new value (constant v) in the proper scope, that indeed lies in the current frame. The reduction results in the empty statement and an updated current frame.

The STMT_ASS_STR rule handles the case where the field of a struct is assigned to; specifically, it reduces an *lval* with field access to f in the assignment to a *lval* without field access, and the value v to be assigned to v', where v' is the value of *lval* (considered as an expression) looked up (**lookup_lval**) in the current stack frame ε, with field f set to v. Accordingly, for a *lval* with m nested field accesses, this rule will have to be applied m times in order for STMT_ASS_V to be used on the result.

Once the expression in the **return** statement has been reduced to a constant, the rule STMT_RET can be applied. The global scope γ_G is always stored at the bottom (index zero) in the stack of scopes, i.e. $\varepsilon[0]$. It is fetched and concatenated with the frame of the most recent caller that is stored on top of the call stack E. Thus, this concatenation will yield a new frame ε that has the same shape as the one before the function being called (of course before reaching the **return** statement some variable mappings in the global scope γ_G could have been changed during function evaluation). The status is also changed to **Return** v which allows for applying E_FUNC_RET.

The STMT_COND2 and STMT_COND3 rules are the standard ones for conditional statements.

[4]Rules for reducing expressions in all contexts are found in Appendix B

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

$$\frac{\begin{array}{c} i = length(\varepsilon) \\ \gamma = \varepsilon[i] \\ \varepsilon' = (i \mapsto (x \mapsto ?)\gamma)\varepsilon \end{array}}{[\mathbf{decl}\, x\, t]((\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{R})} \text{ STMT\_DECL}$$

$$\frac{\varepsilon' = \varepsilon[x \longmapsto v]}{[x := v]((\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{R})} \text{ STMT\_ASS\_V}$$

$$\frac{\begin{array}{c} \mathbf{struct}\,\{x_1 = v_1; \ldots; x_n = v_n\} = \mathbf{lookup\_lval}\,(\varepsilon, lval) \\ v'' = (\mathbf{struct}\,\{x_1 = v_1; \ldots; x_n = v_n\}\,\mathbf{with}\, f := v) \end{array}}{[(lval.f) := v]((\varepsilon, E), \mathbf{R}) \to [lval := v'']((\varepsilon, E), \mathbf{R})} \text{ STMT\_ASS\_STR}$$

$$\frac{\begin{array}{l} \gamma_G = \varepsilon[0] \\ (\varepsilon', f) :: E' = E \\ (stmt, (x_1, d_1), .., (x_n, d_n)) = F(f) \\ \varepsilon'' = (\varepsilon') + +([\gamma_G]) \\ \varepsilon''' = \text{FOLD}(\lambda\, \overline{\varepsilon}\, i.\, \text{if}[d_1, .., d_n][i] \in \{\downarrow, \circ\}\, \text{then}\, \overline{\varepsilon} \\ \text{else}\, \overline{\varepsilon}\, [a \longmapsto v] where\ (v, a) = \text{lookup}_{\text{t}}(\varepsilon, [x_1, ..., x_n][i]))(\varepsilon'')[1...n] \end{array}}{[\mathbf{return}\, v]((\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}]((\varepsilon''', E'), \mathbf{Ret}\ v)} \text{ STMT\_RET}$$

$$\frac{}{[\mathbf{if\ true\ then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{R}) \to [stmt_1](\sigma, \mathbf{R})} \text{ STMT\_COND2}$$

$$\frac{}{[\mathbf{if\ false\ then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{R}) \to [stmt_2](\sigma, \mathbf{R})} \text{ STMT\_COND3}$$

Figure 10: P4 Statement Execution Semantics

The STMT_SEQ1 and STMT_SEQ2 rules are pretty standard. The STMT_SEQ3 rule is used in the situation when the **return** statement does not occur at the end of the the function body. The next statement to reduce will be empty and the status will be changed to **Return**, which is handled by STMT_RET.

The {} brackets indicate a block, while the [ ] brackets indicate a block in progress of being executed. The STMT_BLOCK_ENTER rule is used to enter a block, which entails a new empty scope $\gamma_\emptyset$ being pushed onto the current frame ε, and then the {} brackets are switched to the in-progress ones [ ] to signify that the block is currently being executed. The STMT_BLOCK_EXEC rule simply describes small-step reduction of the block contents, and the STMT_BLOCK_EXIT rule is used in the case where the end of a block is reached, i.e. whenever a block contains only an empty statement: it pops the scope corresponding to the block (the most recent one) from the frame ε.

2.4 Parser

The parser is a part of the P4 language which is generally used to parse packets from bit-string representations to structures of parsed headers, described in Section 13 of the P4 specification. It

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

$$\frac{[stmt_1](\sigma, \mathbf{R}) \to [stmt_1'](\sigma', \mathbf{R})}{[stmt_1; stmt_2](\sigma, \mathbf{R}) \to [stmt_1'; stmt_2](\sigma', \mathbf{R})} \quad \text{STMT\_SEQ1}$$

$$\frac{}{[\emptyset_{\text{stmt}}; stmt](\sigma, \mathbf{R}) \to [stmt](\sigma, \mathbf{R})} \quad \text{STMT\_SEQ2}$$

$$\frac{[stmt_1](\sigma, \mathbf{R}) \to [stmt_1'](\sigma', \mathbf{Ret}\ v)}{[stmt_1; stmt_2](\sigma, \mathbf{R}) \to [\emptyset_{\text{stmt}}](\sigma', \mathbf{Ret}\ v)} \quad \text{STMT\_SEQ3}$$

$$\frac{\varepsilon' = \gamma_\emptyset :: \varepsilon}{[\{stmt\}]((\varepsilon, E), \mathbf{R}) \to [[stmt]]((\varepsilon', E), \mathbf{R})} \quad \text{STMT\_BLOCK\_ENTER}$$

$$\frac{[stmt](\sigma, \mathbf{R}) \to [stmt'](\sigma', \mathbf{R})}{[[stmt]](\sigma, \mathbf{R}) \to [[stmt']](\sigma', \mathbf{R})} \quad \text{STMT\_BLOCK\_EXEC}$$

$$\frac{\varepsilon' = \text{tl}(\varepsilon)}{[[\emptyset_{\text{stmt}}]]((\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}]((\varepsilon', E), \mathbf{R})} \quad \text{STMT\_BLOCK\_EXIT}$$

Figure 11: P4 Statement Execution Semantics: Structural Rules

can be thought of as describing a state machine with three unique states: a *start* state, an **Accept** state and a **Reject** *msg* state. A parser state p (including *start*, but not the abstract final states of **Accept** and **Reject** *msg*) consists of a list of statements to be executed, with a transition statement at the end which decides the parser state to jump to next.

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

$$\frac{}{[\mathbf{verify\ true}\ (\mathbf{errmsg}\ x)](\sigma, \mathbf{R}) \to [\emptyset_{\text{stmt}}](\sigma', \mathbf{R})} \quad \text{STMT\_VERIFY\_3}$$

$$\frac{}{[\mathbf{verify\ false}\ (\mathbf{errmsg}\ x)](\sigma, \mathbf{R}) \to [\emptyset_{\text{stmt}}](\sigma', \mathbf{Reject}\ x)} \quad \text{STMT\_VERIFY\_4}$$

$$\frac{}{[\mathbf{transition}\ p](\sigma, \mathbf{R}) \to [\emptyset_{\text{stmt}}](\sigma, p)} \quad \text{STMT\_TRANSITION}$$

Figure 12: P4 Parser-Specific Statement Execution Semantics

The parser-specific statement semantics is shown in Figure 12. The STMT_VERIFY_3 and STMT_VERIFY_4 rules describe the semantics of **verify**, the expressions having been reduced to values. If the condition holds, the reduction is to the empty statement (i.e. nothing happens and execution continues). If the condition does not hold, reduction is also to the empty statement, but state status is set to **Reject** x. The STMT_TRANSITION rule describes reduction of the **transition** statement, whose only effect on the state is to set status to p (the PARS_STATE or PARS_T_FIN rules can then be used next)

The parser state machine semantics is shown in Figure 13. Note the separate judgment form for the final step of the parser, which goes to a state with status **Accept** or **Reject** (representing

$\boxed{[stmt]s \longrightarrow [stmt']s'}$ parser semantics

$$\frac{[stmt](\sigma, \mathbf{R}) \to [stmt'](\sigma', \mathbf{R})}{[stmt](\sigma, \mathbf{R}) \longrightarrow [stmt'](\sigma', \mathbf{R})} \quad \text{PARS\_STMT}$$

$$\frac{\begin{array}{l}[stmt](\sigma, \mathbf{R}) \to [stmt'](\sigma', \mathbf{Trans}\, x) \\ stmt'' = P(x)\end{array}}{[stmt](\sigma, \mathbf{R}) \longrightarrow [stmt''](\sigma', \mathbf{R})} \quad \text{PARS\_STATE}$$

$\boxed{[stmt]s \longrightarrow s'}$ parser semantics, final step

$$\frac{[stmt](\sigma, \mathbf{R}) \to [stmt'](\sigma, p_{\text{fin}})}{[stmt](\sigma, \mathbf{R}) \longrightarrow (\sigma, p_{\text{fin}})} \quad \text{PARS\_T\_FIN}$$

$$\frac{\begin{array}{l}[stmt](\sigma, \mathbf{R}) \to [\emptyset_{\text{stmt}}](\sigma', \mathbf{R}) \\ x = \text{ParserStateEnd}\end{array}}{[stmt](\sigma, \mathbf{R}) \longrightarrow (\sigma', \mathbf{Reject}\, x)} \quad \text{PARS\_T\_EMPTY}$$

Figure 13: Parser Execution Semantics

the abstract accepting and rejecting states).

The PARS_STMT rule performs a single small-step reduction of the current statement (the body of the current parser state), while the PARS_STATE rule governs transition to the next parser state: if the current statement *stmt* is reduced to *stmt*′ with the status being **Trans** x, the next statement is the body of the parser state with name x, obtained from the map P from parser state names to parser bodies.

The PARS_T_FIN rule says that when reduction using the statement semantics of the current statement results in a status with a final parser state p_{fin}, this is also set as the status in the parser semantics. The PARS_T_EMPTY rule covers the special case when the statement semantics runs out of statements in a parser state, in which case the status is set to **Reject** ParserStateEnd.

2.5 Control Blocks

The control block is a part of the P4 language which is generally used to decide which actions to take (typically forwarding) based on the metadata (headers) which was extracted by the parser, as described in Section 12 of the P4 specification. The two main components of a control block is the match-action table and the actions themselves. Note that part of the functionality is separated into the control plane, which is interfaced with here using the ctrl(*table_name*, v, *m_kind*) function that takes a table name, constant value and matching kind and obtains an action name f and a list of function arguments $v_1, ..., v_n$. Actions can be thought of roughly as functions with no return values. The action can be called implicitly from the match-action process (i.e. in the table application), or explicitly from another action or a control block, as described in Section 13.1.1 of the P4 specification.

The APPLY_TABLE_E rule performs small-step evaluation of the header expression used for the matching.

The APPLY_TABLE_V looks up the table name in the table name map, then uses the result

together with the header to be looked up to obtain an action (together with action arguments) from the control plane.

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[\mathbf{apply}\, table\_name\, e](\sigma, \mathbf{R}) \rightarrow [\mathbf{apply}\, table\_name\, e'](\sigma', \mathbf{R})} \quad \text{STMT\_APPLY\_TABLE\_E}$$

$$\frac{\begin{array}{l}\text{t\_map}\, \mathbf{table\_name} = (e', match\_kind) \\ \text{ctrl}\,(table\_name,\, v,\, match\_kind) = (f, (v_1, .., v_n))\end{array}}{[\mathbf{apply}\, table\_name\, v](\sigma, \mathbf{R}) \rightarrow [\mathbf{null} := (\mathbf{call}\, f(v_1, .., v_n))](\sigma', \mathbf{R})} \quad \text{STMT\_APPLY\_TABLE\_V}$$

Figure 14: Match Action Execution Semantics

A Concrete Syntax of Operations

$\ominus$	::=		
	|	!	boolean negation
	|	¬	bitwise complement
	|	−	signed negation
	|	+	unary plus

Figure 15: P4 Unary Operations

The unary expressions included are shown in Figure 15. 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).

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

Figure 16: P4 Binary Operations

The binary expressions included are shown in Figure 15. These include all of the binary operations in P4.

B Semantics of Expression Reduction

This appendix describes semantics for reducing expressions in certain contexts. The expression semantics are shown in Figure 17. The statement semantics are shown in Figure 18.

The E_FUNC_CALL_ARGS 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 step, after which the corresponding index in the original list of arguments is update with the resulting expression.

8.1 of the P4 specification states that expressions are evaluated left-to-right. Accordingly, the rules for binary operations - E_BINOP1 and E_BINOP2 - are split up so that reduction of the second operand requires that the first operand has been completely reduced to a constant. This is trivial for unary operations (E_UNOP).

$\boxed{[e](\sigma) \rightsquigarrow [e'](\sigma')}$ expression semantics

$$\frac{\begin{array}{l}(stmt, (x_1, d_1), .., (x_n, d_n)) = F(f) \\ i = \min\{j.\ [d_1, .., d_n][j] \in \{\circ, \downarrow\} \wedge \neg(\text{is\_const}\,[e_1, .., e_n][j])\} \\ e = [e_1, .., e_n][i] \\ [e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R}) \\ [e'_1, .., e'_n] = (i \mapsto e')[e_1, .., e_n]\end{array}}{[\mathbf{call}\, f(e_1, .., e_n)](\sigma, \mathbf{R}) \rightsquigarrow [\mathbf{call}\, f(e'_1, .., e'_n)](\sigma', \mathbf{R})}\ \text{E\_FUNC\_CALL\_ARGS}$$

$$\frac{[e'](\sigma, \mathbf{R}) \rightsquigarrow [e''](\sigma', \mathbf{R})}{[e.e'](\sigma, \mathbf{R}) \rightsquigarrow [e.e''](\sigma', \mathbf{R})}\ \text{E\_ACC\_ARG2}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[e.x](\sigma, \mathbf{R}) \rightsquigarrow [e'.x](\sigma', \mathbf{R})}\ \text{E\_ACC\_ARG1}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[\ominus e](\sigma, \mathbf{R}) \rightsquigarrow [\ominus e'](\sigma', \mathbf{R})}\ \text{E\_UNOP\_ARG}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e''](\sigma', \mathbf{R})}{[e \oplus e'](\sigma, \mathbf{R}) \rightsquigarrow [e'' \oplus e'](\sigma', \mathbf{R})}\ \text{E\_BINOP\_ARG1}$$

$$\frac{[e'](\sigma, \mathbf{R}) \rightsquigarrow [e''](\sigma', \mathbf{R})}{[v \oplus e'](\sigma, \mathbf{R}) \rightsquigarrow [e \oplus e''](\sigma', \mathbf{R})}\ \text{E\_BINOP\_ARG2}$$

Figure 17: P4 Expression Evaluation Semantics

$\boxed{[stmt]s \rightarrow [stmt']s'}$ statement semantics

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[\mathbf{return}\ e](\sigma, \mathbf{R}) \rightarrow [\mathbf{return}\ e'](\sigma', \mathbf{R})}\ \text{STMT\_RET\_E}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[x := e](\sigma, \mathbf{R}) \rightarrow [x := e'](\sigma', \mathbf{R})}\ \text{STMT\_ASS\_E}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[\mathbf{if}\ e\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma, \mathbf{R}) \rightarrow [\mathbf{if}\ e'\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2](\sigma', \mathbf{R})}\ \text{STMT\_COND\_E}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e''](\sigma', \mathbf{R})}{[\mathbf{verify}\ e\ e'](\sigma, \mathbf{R}) \rightarrow [\mathbf{verify}\ e''\ e'](\sigma', \mathbf{R})}\ \text{STMT\_VERIFY\_E1}$$

$$\frac{[e](\sigma, \mathbf{R}) \rightsquigarrow [e'](\sigma', \mathbf{R})}{[\mathbf{verify}\ b\ e](\sigma, \mathbf{R}) \rightarrow [\mathbf{verify}\ b\ e'](\sigma', \mathbf{R})}\ \text{STMT\_VERIFY\_E2}$$

Figure 18: P4 Statement Execution Semantics