# The p4ott P4 Formalization

Anoud Alshnakat
Didrik Lundberg

March 1, 2022

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

| | |
|---|---|
| $x$, $f$, $a$, $tbl$ | string |
| $b$ | boolean |
| $bl$ | bit-string |
| $i$ | 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 $a$, $x$ (variable or parser state name), $msg$ (error message), $table\_name$ (match-action table name) or $f$ (function or field name). $bl$ is a list of Boolean values, used to represent bit-strings of fixed width.

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 |
| | \| | $\text{bool}_\text{t}$ | |
| | \| | $\text{bit}_\text{t}$ | |

| | | | |
|---|---|---|---|
| $t$ | ::= | | types |
| | \| | $bt$ | |
| | \| | $\text{struct}_\text{t}\ t_1, \ldots, t_n$ | |
| | \| | $\text{header}_\text{t}\ t_1, \ldots, t_n$ | |

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 |
| | \| | **var** $x$ | variable or extern object |
| | \| | $\{e_1, .., e_n\}$ | expression list |
| | \| | $e.e'$ | field access |
| | \| | $\ominus e$ | unary operation |
| | \| | $e_1 \oplus e_2$ | binary operation |
| | \| | $e_1 + + e_2$ | concatenation of bit-strings |
| | \| | $e_1[e_2 : e_3]$ | bit-slice |
| | \| | **call** $f(e_1, .., e_n)$ | function call |
| | \| | **exec** $stmt$ | function execution |
| | \| | **call** $lval\ f(e_1, .., e_n)$ | extern method call |
| | \| | **select** $e\{v_1 : x_1; ...; v_n : x_n\}x$ | select |
| | \| | $(e)$ | S |

Figure 3: P4 Expressions

First, an expression can be a value: a Boolean or an integer (collectively referred to as constant values $v$), or a variable or parser state name $x$. Lists of expressions can be used in declarations of variables of struct types. The fields of these structs may be accessed, which is denoted in the usual manner. 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.

The **select** expression is similar to a switch statement in C or Java. The expression $e$ is evaluated, and then matched against $v_1, \ldots, v_n$. If some match is successful, the **select** expression evaluates to the string at the corresponding index. If no match occurs, then it instead evaluates to the default string $x$.

[1]The concrete syntax of the many 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 expression $e$ 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, the name of which is the result of evaluating $e$. The **apply** statement applies the match-action table with name *table_name* (found in the control plane) to the result of evaluating $e$, thus obtaining an action (modeled as a function call) to execute next.

| | | | |
|---|---|---|---|
| *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** $e$ | transition |
| | \| | **apply** $tbl\ e$ | apply |

Figure 4: P4 Statements

The assignment can assign to *lval*s (shown in Figure 5), which include variables identified by their names, a null variable (used to model method calls) 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 16. 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 $\sigma$ and the state status $t$. The state memory $\sigma$ consists of a tuple $(\varepsilon,\ E)$, where $\varepsilon$ is a stack of scopes $\gamma$ (induced by entering nested blocks) which hold the values of variables which are currently visible, and $E$ holds variable mappings which belong to previous caller contexts.

| | | | |
|---|---|---|---|
| $t$ | ::= | | execution status |
| | \| | $\mathbf{R}$ | |
| | \| | $p$ | |
| | \| | $\bot$ | |
| $s$ | ::= | | execution state |
| | \| | $(C, \sigma, t)$ $\quad \mathsf{M}$ | |

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 $\gamma$:

- $\mathrm{dom}(\gamma)$: Gets the domain of $\gamma$: obtains the set of variable names $x \in X$ which are mapped to values in $\gamma$.
- $(x \mapsto v)\ \gamma$: Updates a variable mapping in $\gamma$: yields the scope $\gamma'$, which is just $\gamma$ 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 $\varepsilon$ is a stack of scopes where the global scope $\gamma_G$ is located at the bottom; that is, in location $\varepsilon[0]$. When a frame is considered a list the head of the list (i.e $\varepsilon[0]$) represent the bottom of the stack. The current scope - that which was most recently entered by execution - is stored on the top of $\varepsilon$ (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 $\varepsilon$.

The following operations can be performed on a frame $\varepsilon$:

- $\gamma :: \varepsilon$: Add a scope $\gamma$ on bottom of $\varepsilon$ (i.e. cons).
- $(i \mapsto \gamma)\ \varepsilon$: Updates the scope located at index $i$ of $\varepsilon$ by setting it to $\gamma$.

The call stack $E$ is a stack of frames used whenever a function call occurs. When a function call is executed, the frame $\varepsilon$ (minus the global scope $\gamma_G$) of the caller will be pushed onto $E$. When the callee function finishes execution and returns, $\varepsilon$ will be popped from $E$ and pushed onto a frame containing only $\gamma_G$. Note that this means that the same $\gamma_G$ 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 $\varepsilon$ onto the call stack $E$.

The status $\mathbf{R}$ represents that the program is executing under regular circumstances. $\mathbf{Ret}\ v$ 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 ($p_{\text{fin}}$) 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 table map $Tb$ mapping table names to tuples of expressions and matching kinds, 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

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

In the E_LOOKUP rule, the lookup function ensures that the variable name $x$ is evaluated in the uppermost (i.e. most recently entered) scope of $\varepsilon$ 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 (for non-out directions) or variables (directions with out). Arguments with out-directions are looked up in the caller frame, and a tuple of the resulting value and the originating variable is assigned to a variable with the corresponding parameter name in a fresh callee scope $\gamma'$. Similarly, arguments with non-out directions are simply assigned to their parameter names in $\gamma'$. Finally, $\gamma'$ is put on top of the global scope $\gamma_G$ in order to form the new frame of the callee $\varepsilon'$. The old current frame $\varepsilon$ (minus $\gamma_G$) 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 and directions). 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 **Ret** $v$ to $v$. This also changes the status to **R**.

The E_S_ACC rule is used to access the values of fields in structs, and the E_H_ACC rule is similarly used for headers.

The E_SEL_ACC rule is used to match the given value $v$ against the key-value list, in the case a match exists. The E_SEL_DEF rule is used for the default case, when no match exists.

[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

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

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

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

$$\frac{\begin{array}{l}\textbf{not\_empty}\, stmt \\ ctx \vdash [stmt](C, \sigma, \mathbf{R}) \rightarrow [stmt'](C', \sigma', t)\end{array}}{ctx \vdash [\textbf{exec}\, stmt](C, \sigma, \mathbf{R}) \rightsquigarrow [\textbf{exec}\, stmt'](C', \sigma', t)} \quad \text{E\_FUNC\_EXEC}$$

$$\frac{\begin{array}{l}(\varepsilon', f) :: E' = E \\ (stmt, (x_1, d_1), .., (x_n, d_n)) = F(f) \\ \gamma_G = \varepsilon[0] \\ \varepsilon'' = (\gamma_G) :: \varepsilon' \\ \varepsilon''' = \text{FOLD}(\lambda\, \overline{\varepsilon}\, i.\, \text{if}[d_1, .., d_n][i] \in \{\downarrow, \circ\}\, \text{then}\, \overline{\varepsilon} \\ \text{else}\, \overline{\varepsilon}\, [a \longmapsto v] where\ (v, a) = \text{lookup}_\text{t}(\varepsilon, [x_1, ..., x_n][i]))(\varepsilon'')[1...n]\end{array}}{(Ty, X, F, Tb) \vdash [\textbf{exec}\, \textbf{return}\, v](C, (\varepsilon, E), \mathbf{R}) \rightsquigarrow [v](C, (\varepsilon''', E'), \mathbf{R})} \quad \text{E\_FUNC\_RET}$$

$$\frac{v = \textbf{struct}\, \{f_1 = v_1; \, ...\, ; f_n = v_n\}(f)}{ctx \vdash [\textbf{struct}\, \{f_1 = v_1; \, ...\, ; f_n = v_n\}.f](C, \sigma, \mathbf{R}) \rightsquigarrow [v](C, \sigma, \mathbf{R})} \quad \text{E\_S\_ACC}$$

$$\frac{v = \textbf{header}\, boolv\{f_1 = v_1; \, ...\, ; f_n = v_n\}(f)}{ctx \vdash [\textbf{header}\, boolv\{f_1 = v_1; \, ...\, ; f_n = v_n\}.f](C, \sigma, \mathbf{R}) \rightsquigarrow [v](C, \sigma, \mathbf{R})} \quad \text{E\_H\_ACC}$$

$$\frac{x' = \{v_1 : x_1; ...; v_n : x_n; \_ : x\}(v)}{ctx \vdash [\textbf{select}\, v\{v_1 : x_1; \, ...\, ; v_n : x_n\}x](C, \sigma, \mathbf{R}) \rightsquigarrow [x'](C, \sigma, \mathbf{R})} \quad \text{E\_SEL\_ACC}$$

$$\frac{v \notin \text{dom}(\{v_1 : x_1; ...; v_n : x_n\})}{ctx \vdash [\textbf{select}\, v\{v_1 : x_1; \, ...\, ; v_n : x_n\}x](C, \sigma, \mathbf{R}) \rightsquigarrow [x](C, \sigma, \mathbf{R})} \quad \text{E\_SEL\_DEF}$$

Figure 7: P4 Expression Evaluation Semantics

## 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 ($\varepsilon$, 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 $\varepsilon$. 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 $\varepsilon$.
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 $\varepsilon$.

$$
\begin{array}{lll}
d & ::= & \text{parameter direction} \\
 & | & \downarrow \\
 & | & \uparrow \\
 & | & \updownarrow \\
 & | & \circ
\end{array}
$$

Figure 8: direction syntax

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

Figure 9: called function name list

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

$$\frac{\begin{array}{c} i = length(\varepsilon) - 1 \\ \gamma = \varepsilon[i] \\ v = \text{arb\_from\_t}t \\ \varepsilon' = (i \mapsto (x \mapsto v)\gamma)\varepsilon \end{array}}{ctx \vdash [\mathbf{decl}\, x\, t](C, (\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}](C, (\varepsilon', E), \mathbf{R})} \text{ STMT\_DECL}$$

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

$$\frac{}{ctx \vdash [\mathbf{null} := v](C, (\varepsilon, E), \mathbf{R}) \to [\emptyset_{\text{stmt}}](C, (\varepsilon, E), \mathbf{R})} \text{ STMT\_ASS\_NULL}$$

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

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

Figure 10: P4 Statement Execution Semantics

## 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 current frame, and never in $E$. The antecedent $\varepsilon[x \longmapsto v]$ obtains the topmost (i.e. most recently entered) scope in the current frame $\varepsilon$ containing the variable, then updates the mapping of the variable name $x$ to the new value (constant $v$) in this scope. The

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

reduction results in the empty statement and an updated current frame.

The STMT_ASS_NULL rule handles the case where null variable is assigned to, i.e., in method calls.

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

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

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

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

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

$$\frac{ctx \vdash [stmt](C, \sigma, t) \to [stmt'](C', \sigma', t')}{ctx \vdash [[stmt]](C, \sigma, t) \to [[stmt']](C', \sigma', t')} \text{ STMT\_BLOCK\_EXEC}$$

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

Figure 11: P4 Statement Execution Semantics: Structural Rules

The STMT_SEQ1 and STMT_SEQ2 rules are pretty standard.

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 $\varepsilon$, 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 $\varepsilon$.

## 2.4 Parser Block Semantics

The parser programmable block is a part of the P4 pipeline 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 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.

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

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

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

$$\frac{\begin{array}{c}x' = \text{“}parseError\text{”}\\ x'' = \text{“}reject\text{”}\end{array}}{ctx \vdash [\mathbf{verify}\,\mathbf{false}\,(\mathbf{errmsg}\,x)](C,\sigma,\mathbf{R}) \to [x' := (\mathbf{errmsg}\,x); \mathbf{transition}\,x'](C,\sigma,\mathbf{R})}\ \text{STMT\_VERIFY\_4}$$

$$\frac{\mathbf{not\_final\_state}\,(x)}{ctx \vdash [\mathbf{transition}\,x](C,\sigma,\mathbf{R}) \to [\emptyset_{\text{stmt}}](C,\sigma,\mathbf{Trans}\,x)}\ \text{STMT\_TRANS\_1}$$

$$\frac{x = \text{“}accept\text{”}}{ctx \vdash [\mathbf{transition}\,x](C,\sigma,\mathbf{R}) \to [\emptyset_{\text{stmt}}](C,\sigma,\mathbf{Accept})}\ \text{STMT\_TRANS\_2}$$

$$\frac{x = \text{“}reject\text{”}}{ctx \vdash [\mathbf{transition}\,x](C,\sigma,\mathbf{R}) \to [\emptyset_{\text{stmt}}](C,\sigma,\mathbf{Reject})}\ \text{STMT\_TRANS\_3}$$

Figure 12: P4 Parser-Specific Statement Execution Semantics

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_TRANS rules describe reduction of the **transition** statement, whose only effect on the state is to set status to indicate next parser state (the PARS_STATE or PARS_T_FIN rules can then be used next)

$\boxed{ctx_P \vdash [stmt]s \longrightarrow_p [stmt']s'}$ parser block semantics

$$\frac{\begin{array}{c}t \neq pars\_fin\\ (Ty,X,F,\mathbf{empty}) \vdash [stmt]s \to [stmt']s'\end{array}}{(Ty,X,F,P) \vdash [stmt]s \longrightarrow_p [stmt']s'}\ \text{PARS\_STMT}$$

$$\frac{\begin{array}{c}(Ty,X,F,\mathbf{empty}) \vdash [stmt](C,\sigma,\mathbf{R}) \to [stmt'](C',\sigma',\mathbf{Trans}\,x)\\ stmt'' = P(x)\end{array}}{(Ty,X,F,P) \vdash [stmt](C,\sigma,\mathbf{R}) \longrightarrow_p [stmt''](C',\sigma',\mathbf{R})}\ \text{PARS\_STATE}$$

$$\frac{\begin{array}{c}(Ty,X,F,\mathbf{empty}) \vdash [stmt](C,\sigma,\mathbf{R}) \to [\emptyset_{\text{stmt}}](C',\sigma',\mathbf{R})\\ x = \text{“}reject\text{”}\end{array}}{(Ty,X,F,P) \vdash [stmt](C,\sigma,\mathbf{R}) \longrightarrow_p [\mathbf{transition}\,x](C,\sigma',\mathbf{R})}\ \text{PARS\_EMPTY}$$

$$\frac{(Ty,X,F,\mathbf{empty}) \vdash [stmt](C,\sigma,\mathbf{R}) \to [stmt'](C',\sigma',p_{\text{fin}})}{(Ty,X,F,P) \vdash [stmt](C,\sigma,\mathbf{R}) \longrightarrow_p [stmt'](C',\sigma',p_{\text{fin}})}\ \text{PARS\_FIN}$$

Figure 13: Parser Block-Level Semantics

The parser state machine semantics is shown in Figure 13.

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 $p_{\text{fin}}$, 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 Block Semantics

The control block is a part of the P4 pipeline 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 are the match-action tables 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.

$$\frac{ctx \vdash [e]s \rightsquigarrow [e']s'}{ctx \vdash [\textbf{apply}\, tbl\ e]s \rightarrow [\textbf{apply}\, tbl\ e']s'}\ \text{STMT\_APPLY\_TABLE\_E}$$

$$\frac{\begin{array}{l}\text{t\_map}\, tbl\ =\ (e', mk)\\ \text{ctrl}\,(tbl,\ v,\ mk\,) = (f, (v_1, .., v_n))\end{array}}{(Ty, X, F, Tb) \vdash [\textbf{apply}\, tbl\ v](C, \sigma, \mathbf{R}) \rightarrow [\textbf{null} := (\textbf{call}\, f(v_1, .., v_n))](C, \sigma, \mathbf{R})}\ \text{STMT\_APPLY\_TABLE\_V}$$

Figure 14: Match Action Execution Semantics

The statement semantics unique to the control block is shown in Figure 14. Note that the body of the control block consists of the statements inside the apply block, and that executing a return statement at this level signifies return from the control block as a whole. The block-level semantics of the control block are shown in Figure 15. It's almost the same as the statement semantics, the only difference being that the return statement causes an immediate reduction to the empty statement.

## 2.6 Architecture-Level Semantics

The architecture-level semantics is the topmost-level semantics, and it describes the entirety of the P4 pipeline from input packets to output packets. The judgment form, shown in Figure 17, consists

$\boxed{ctx_C \vdash [stmt]s \longrightarrow_c [stmt']s'}$ control block semantics

$$\frac{stmt \neq return \quad (Ty, X, F, Tb) \vdash [stmt]s \rightarrow [stmt']s'}{(Ty, X, F, Tb) \vdash [stmt]s \longrightarrow_c [stmt']s'} \text{ CTRL\_STMT}$$

$$\frac{}{ctx_C \vdash [\textbf{return}\ v]s \longrightarrow_c [\emptyset_{\text{stmt}}]s} \text{ CTRL\_RET}$$

$$\frac{}{ctx_C \vdash [\textbf{return}\ v; stmt]s \longrightarrow_c [\emptyset_{\text{stmt}}]s} \text{ CTRL\_RET\_2}$$

Figure 15: Control Block-Level Semantics

of multiple components: starting from the left, an *architectural context* $ctx_A$ on the left-hand side of the turnstile, which contains the following:

1. The *architectural block list* $\overline{ab}$: the architectural block represents one of the four fundamental stages in a P4-compatible architecture. First, there are the input (**inp**) and output (**out**) stages: these stages are just translators between the P4 input packet format and the general I/O format consisting of a list of ports, each with pending packets to be arbitrated/sent off. The demux block functionality may be modeled as part of the output stage, or it may be its own fixed-function block preceding the output stage. Second, there are the programmable blocks - parser blocks and control blocks, which are described in more detail elsewhere in this document, as well as the fixed-function blocks. Fixed-function blocks can be thought of as analoguous to extern calls, but on the architectural level. They perform the parts of the functionality of the P4-programmable network element that is not written in the P4 language.

2. The *programmable block map* $B_p$, which is a map between names of programmable blocks (strings) and the necessary information required to execute the block in question. For parser blocks, this is the list of directed parameters and the parser state map $P$ between parser state names and their bodies (statements). For programmable blocks, this is the body of the control block (found in the topmost apply statement), the list of directed parameters and a table map $Tb$ between names of tables and tuples of expressions and matching kinds.

3. The *programmable block map* $B_{ff}$, which is a map between names of fixed-function blocks and their lists of directed parameters, as well as the implementation of the fixed fiunction itself - this is a function between an expression list (arguments), the architectural scope, and the current frame to an updated architectural scope and current frame.

4. The *input function* $f_{in}$ and the output function $f_{out}$, which are both functions from the IO list and the architectural scope to an updates IO list and architectural scope.

5. The *type map* $Ty$, which is a (currently unused) map from names of types to their actual types.

6. The *extern map* $X$, which is a map from extern names (strings) to tuples of extern functions (functions from tuples of lvals, arguments and states to option types of values and states) and their directed parameter lists. Note that that since the state contains the control plane configuration, extern functions may modify it.

7. The *function map* $F$, which is a map from function names (strings) to tuples of function bodies (statements) and their directed parameter lists.

The architectural context contains everything which can not change between reductions.

The left-hand and right-hand sides of the reduction both consist of an *architectural statement* and an *architectural environment* $env_A$, which may change between reductions. The architectural statement is either a regular statement as defined above, or a programmable or fixed-function block call which both take lists of directed parameters and function names. $env_A$has five components:

1. The *architectural block index*: This is an index which informs us of which architectural block in $\overline{ab}$ is currently being reduced.
2. The *in-progress flag*, which is a Boolean telling us whether a programmable block is currently being executed (as opposed to returning or being initialised) or not.
3. The *input list*, which is a list of incoming packets (represented as tuples of lists of Booleans and numbers signifying input ports)
4. The *output list*, which is the same as the input list, but used for output.
5. The *architectural scope*, which is of the same type as the regular scope, but used for storing values in-between non-adjacent blocks.

The rules of the architecture-level semantics can be divided into several categories:

1. The ARCH_IN, ARCH_PBL_CALL, ARCH_FFBL_CALL and ARCH_OUT rules all deal with queueing the next block to be executed. Input and output takes place immediately by applying their respective functions.
2. The ARCH_PARSER_INIT and ARCH_CONTROL_INIT rules are similar to the E_FUNC_CALL_NEWFRAME rule in the expression semantics: both set up the values of variables in the new parser and control blocks, respectively. In the parser case, this rule also sets the value of *parseError* to *NoError* and the statement to be executed to the statement mapped to *start* in $P$.
3. The ARCH_PARSER_EXEC, ARCH_CONTROL_EXEC and ARCH_FFBLOCK_EXEC rules all govern the actual execution of blocks. ARCH_FFBLOCK_EXEC is different from the other two in that it performs execution in one step only, and thus also checks for properly reduced arguments the same way ARCH_PARSER_INIT (for example) does. Accordingly, it also increments the architecture block index. ARCH_PARSER_EXEC takes a step in the parser semantics (using the parser map $P$ that was obtained from $B_p$) that does not start in a state with status *pars_fin*. ARCH_CONTROL_EXEC similarly takes a step in the control semantics using the table map $Tb$ that was obtained from $B_p$.
4. The ARCH_PARSER_RET and ARCH_CONTROL_RET rules govern the return phase of programmable block execution. They are similar in many respects to E_FUNC_RET, but they also increment the architecture block index, and ARCH_PARSER_RET furthermore copies back the value of *parseError* to the architecture scope.

$\boxed{ctx_A \vdash [arch\_stmt](env_A, \sigma, t) \longrightarrow_A [arch\_stmt'](env_A', \sigma', t')}$ architecture-level semantics

$$\frac{\begin{array}{l}\mathbf{inp} = \overline{ab}[i] \\ (\overline{io}'', \gamma') = f_{in}(\overline{io}, \gamma) \\ i' = i + 1\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R}) \longrightarrow_A [\emptyset_{\text{stmt}}]((i', \mathbf{false}, \overline{io}'', \overline{io}', \gamma'), C, \sigma, \mathbf{R})}\ \text{ARCH\_IN}$$

$$\frac{pblock\ x(e_1, .., e_n) = \overline{ab}[i]}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R}) \longrightarrow_A [\mathbf{pbl\_call}\, x(e_1, .., e_n)]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R})}\ \text{ARCH\_PBL\_CALL}$$

$$\frac{ffblock\ x(e_1, .., e_n) = \overline{ab}[i]}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R}) \longrightarrow_A [\mathbf{ffbl\_call}\, x(e_1, .., e_n)]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R})}\ \text{ARCH\_FFBL\_CALL}$$

$$\frac{\begin{array}{l}\mathbf{out} = \overline{ab}[i] \\ (\overline{io}'', \gamma') = f_{out}(\overline{io}', \gamma) \\ i' = 0\end{array}}{ctx_A \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, \sigma, \mathbf{R}) \longrightarrow_A [\emptyset_{\text{stmt}}]((i', \mathbf{false}, \overline{io}, \overline{io}'', \gamma'), C, \sigma, \mathbf{R})}\ \text{ARCH\_OUT}$$

$$\frac{\begin{array}{l}\mathbf{parser}\,((x_1, d_1), .., (x_n, d_n))P = B_p(f) \\ x = \text{“}start\text{”} \\ stmt = P(x) \\ \forall d, e, i \leqslant n.\, d = [d_1, .., d_n][i] \wedge e = [e_1, .., e_n][i] \implies ((d \in \{\circ, \downarrow\} \implies \text{is\_const}\, e) \wedge (d \in \{\updownarrow, \uparrow\} \implies \text{is\_var}\, e)) \\ \forall d, e, x, i \leqslant n.\, x = [x_1, ..., x_n][i] \wedge e = [e_1, ..., e_n][i] \wedge d = [d_1, ..., d_n][i] \implies \gamma'(x) = \begin{cases}(\text{lookup}_{\text{v}}(\varepsilon, e), e) & \text{if } d \in \{\uparrow, \updownarrow\} \\ (e, \perp) & \text{if } d \in \{\downarrow, \circ\}\end{cases} \\ \gamma_G = \varepsilon[0] \\ \varepsilon' = [\gamma_G] + +[\gamma'] \\ v = NoError \\ lval = parseError \\ \varepsilon'' = \varepsilon'[lval \longmapsto v]\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\mathbf{pbl\_call}\, f(e_1, .., e_n)]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon, E), \mathbf{R}) \longrightarrow_A [stmt]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon', E), \mathbf{R})}\ \text{ARCH\_PARSER\_INIT}$$

$$\frac{\begin{array}{l}\mathbf{control}\, stmt((x_1, d_1), .., (x_n, d_n))\, Tb = B_p(f) \\ \forall d, e, i \leqslant n.\, d = [d_1, .., d_n][i] \wedge e = [e_1, .., e_n][i] \implies ((d \in \{\circ, \downarrow\} \implies \text{is\_const}\, e) \wedge (d \in \{\updownarrow, \uparrow\} \implies \text{is\_var}\, e)) \\ \forall d, e, x, i \leqslant n.\, x = [x_1, ..., x_n][i] \wedge e = [e_1, ..., e_n][i] \wedge d = [d_1, ..., d_n][i] \implies \gamma''(x) = \begin{cases}(\text{lookup}_{\text{v}}(\varepsilon, e), e) & \text{if } d \in \{\uparrow, \updownarrow\} \\ (e, \perp) & \text{if } d \in \{\downarrow, \circ\}\end{cases} \\ \gamma'_G = \varepsilon[0] \\ \varepsilon' = [\gamma'_G] + +[\gamma'']\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\mathbf{pbl\_call}\, f(e_1, .., e_n)]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon, E), \mathbf{R}) \longrightarrow_A [stmt]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon', E), \mathbf{R})}\ \text{ARCH\_CONTROL\_INIT}$$

$$\frac{\begin{array}{l}pblock\ x(e_1, .., e_n) = \overline{ab}[i] \\ \mathbf{parser}\,((x_1, d_1), .., (x_n, d_n))P = B_p(x) \\ t \neq pars\_fin \\ (Ty, X, F, P) \vdash [stmt](C, \sigma, t) \longrightarrow_p [stmt'](C', \sigma', t')\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [stmt]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, \sigma, t) \longrightarrow_A [stmt']((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C', \sigma', t')}\ \text{ARCH\_PARSER\_EXEC}$$

$$\frac{\begin{array}{l}pblock\ x(e_1, .., e_n) = \overline{ab}[i] \\ \mathbf{control}\, stmt''((x_1, d_1), .., (x_n, d_n))\, Tb = B_p(x) \\ (Ty, X, F, Tb) \vdash [stmt](C, \sigma, t) \longrightarrow_c [stmt'](C', \sigma', t')\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [stmt]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, \sigma, t) \longrightarrow_A [stmt']((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C', \sigma', t')}\ \text{ARCH\_CONTROL\_EXEC}$$

$$\frac{\begin{array}{l}pblock\ f(e_1, .., e_n) = \overline{ab}[i] \\ \mathbf{parser}\,((x_1, d_1), .., (x_n, d_n))P = B_p(f) \\ \gamma'_G = \varepsilon[0] \\ [\gamma''] = \text{FOLD}(\lambda\, \overline{\varepsilon}\, i.\, \text{if}[d_1, .., d_n][i] \in \{\downarrow, \circ\}\, \text{then}\, \overline{\varepsilon} \\ \text{else}\, \overline{\varepsilon}\, [a \longmapsto v] where\ (v, a) = \text{lookup}_{\text{t}}(\varepsilon, [x_1, ..., x_n][i]))([\gamma])[1...n] \\ lval = parseError \\ x = parseError \\ v = \text{lookup}_{\text{v}}(\varepsilon, x) \\ [\gamma'''] = [\gamma''][lval \longmapsto v] \\ i' = i + 1\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon, E), p_{\text{fin}}) \longrightarrow_A [\emptyset_{\text{stmt}}]((i', \mathbf{false}, \overline{io}, \overline{io}', \gamma'''), C, ([\gamma'_G], \emptyset_E), \mathbf{R})}\ \text{ARCH\_PARSER\_RET}$$

$$\frac{\begin{array}{l}pblock\ f(e_1, .., e_n) = \overline{ab}[i] \\ \mathbf{control}\, stmt((x_1, d_1), .., (x_n, d_n))\, Tb = B_p(f) \\ \gamma'_G = \varepsilon[0] \\ [\gamma''] = \text{FOLD}(\lambda\, \overline{\varepsilon}\, i.\, \text{if}[d_1, .., d_n][i] \in \{\downarrow, \circ\}\, \text{then}\, \overline{\varepsilon} \\ \text{else}\, \overline{\varepsilon}\, [a \longmapsto v] where\ (v, a) = \text{lookup}_{\text{t}}(\varepsilon, [x_1, ..., x_n][i]))([\gamma])[1...n] \\ i' = i + 1\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\emptyset_{\text{stmt}}]((i, \mathbf{true}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon, E), t) \longrightarrow_A [\emptyset_{\text{stmt}}]((i', \mathbf{false}, \overline{io}, \overline{io}', \gamma''), C, ([\gamma'_G], \emptyset_E), \mathbf{R})}\ \text{ARCH\_CONTROL\_RET}$$

$$\frac{\begin{array}{l}i' = i + 1 \\ ff((x_1, d_1), .., (x_n, d_n)) = B_{ff}(f) \\ \forall d, e, i \leqslant n.\, d = [d_1, .., d_n][i] \wedge e = [e_1, .., e_n][i] \implies ((d \in \{\circ, \downarrow\} \implies \text{is\_const}\, e) \wedge (d \in \{\updownarrow, \uparrow\} \implies \text{is\_var}\, e)) \\ (\gamma', \varepsilon') = ff((e_1, .., e_n), \gamma, \varepsilon)\end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\mathbf{ffbl\_call}\, f(e_1, .., e_n)]((i, \mathbf{false}, \overline{io}, \overline{io}', \gamma), C, (\varepsilon, E), t) \longrightarrow_A [\emptyset_{\text{stmt}}]((i', \mathbf{false}, \overline{io}, \overline{io}', \gamma'), C, (\varepsilon', E), t')}\ \text{ARCH\_FFBLOCK\_EXEC}$$

Figure 16: Architecture-Level Semantics

# A Concrete Syntax of Operations

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

Figure 17: P4 Unary Operations

The unary expressions included are shown in Figure 18. 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 |
| | \| | $\ll$ | logical left-shift |
| | \| | $\gg$ | logical 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 18: P4 Binary Operations

The binary expressions included are shown in Figure 18. 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 20. The statement semantics are shown in Figure 21. The architecture semantics are shown in Figure 22.

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

$$\frac{\begin{array}{l}(stmt, (x_1, d_1), .., (x_n, d_n)) = F(f)\\ i = \min\{j.\ [d_1, .., d_n][j] \in \{\circ, \downarrow\} \wedge \neg(\text{is\_const}\,[e_1, .., e_n][j])\}\\ e = [e_1, .., e_n][i]\\ (Ty, X, F, Tb) \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t')\\ [e'_1, .., e'_n] = (i \mapsto e')[e_1, .., e_n]\end{array}}{(Ty, X, F, Tb) \vdash [\textbf{call}\, f(e_1, .., e_n)](C, \sigma, t) \rightsquigarrow [\textbf{call}\, f(e'_1, .., e'_n)](C', \sigma', t')}\ \text{E\_FUNC\_CALL\_ARGS}$$

$$\frac{ctx \vdash [e'](C, \sigma, t) \rightsquigarrow [e''](C', \sigma', t')}{ctx \vdash [e.e'](C, \sigma, t) \rightsquigarrow [e.e''](C', \sigma', t')}\ \text{E\_ACC\_ARG2}$$

$$\frac{ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t')}{ctx \vdash [e.x](C, \sigma, t) \rightsquigarrow [e'.x](C', \sigma', t')}\ \text{E\_ACC\_ARG1}$$

$$\frac{ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t')}{ctx \vdash [\textbf{select}\, e\{v_1 : x_1; \ldots; v_n : x_n\}x](C, \sigma, t) \rightsquigarrow [\textbf{select}\, e'\{v_1 : x_1; \ldots; v_n : x_n\}x](C', \sigma', t')}\ \text{E\_SEL\_ARG}$$

$$\frac{ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t')}{ctx \vdash [\ominus e](C, \sigma, t) \rightsquigarrow [\ominus e'](C', \sigma', t')}\ \text{E\_UNOP\_ARG}$$

$$\frac{ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e''](C', \sigma', t')}{ctx \vdash [e \oplus e'](C, \sigma, t) \rightsquigarrow [e'' \oplus e'](C', \sigma', t')}\ \text{E\_BINOP\_ARG1}$$

$$\frac{ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t')}{ctx \vdash [v \oplus e](C, \sigma, t) \rightsquigarrow [v \oplus e'](C', \sigma', t')}\ \text{E\_BINOP\_ARG2}$$

Figure 19: Expression Reduction-of-Argument Semantics

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

$$\frac{ctx \vdash [e]s \rightsquigarrow [e']s'}{ctx \vdash [\textbf{return}\, e]s \rightarrow [\textbf{return}\, e']s'}\ \text{STMT\_RET\_E}$$

$$\frac{ctx \vdash [e]s \rightsquigarrow [e']s'}{ctx \vdash [lval := e]s \rightarrow [lval := e']s'}\ \text{STMT\_ASS\_E}$$

$$\frac{ctx \vdash [e]s \rightsquigarrow [e']s'}{ctx \vdash [\textbf{if}\, e\, \textbf{then}\, stmt_1\, \textbf{else}\, stmt_2]s \rightarrow [\textbf{if}\, e'\, \textbf{then}\, stmt_1\, \textbf{else}\, stmt_2]s'}\ \text{STMT\_COND\_E}$$

$$\frac{ctx \vdash [e]s \rightsquigarrow [e'']s'}{ctx \vdash [\textbf{verify}\, e\, e']s \rightarrow [\textbf{verify}\, e''\, e']s'}\ \text{STMT\_VERIFY\_E1}$$

$$\frac{ctx \vdash [e]s \rightsquigarrow [e']s'}{ctx \vdash [\textbf{verify}\, b\, e]s \rightarrow [\textbf{verify}\, b\, e']s'}\ \text{STMT\_VERIFY\_E2}$$

Figure 20: Statement Reduction-of-Argument Semantics

$\boxed{ctx_A \vdash env_A\, ctx[stmt](\sigma, t) \rightharpoonup env_A{}'\, ctx'[stmt'](\sigma', t')}$ architectural-level semantics

$$\frac{\begin{array}{l} pblock = B_p(f) \\ ((x_1, d_1), .., (x_n, d_n)) = \textbf{args\_of\_pbl}\,(pblock) \\ i = \min\{j.\ [d_1, .., d_n][j] \in \{\circ, \downarrow\} \wedge \neg(\text{is\_const}\,[e_1, .., e_n][j])\} \\ e = [e_1, .., e_n][i'] \\ ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t') \\ [e'_1, .., e'_n] = (i' \mapsto e')[e_1, .., e_n] \end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\textbf{pbl\_call}\, f(e_1, .., e_n)](env_A, C, \sigma, t) \longrightarrow_A [\textbf{pbl\_call}\, f(e'_1, .., e'_n)](env_A, C', \sigma', t')}\ \text{AR}$$

$$\frac{\begin{array}{l} ff((x_1, d_1), .., (x_n, d_n)) = B_{ff}(f) \\ i = \min\{j.\ [d_1, .., d_n][j] \in \{\circ, \downarrow\} \wedge \neg(\text{is\_const}\,[e_1, .., e_n][j])\} \\ e = [e_1, .., e_n][i'] \\ ctx \vdash [e](C, \sigma, t) \rightsquigarrow [e'](C', \sigma', t') \\ [e'_1, .., e'_n] = (i' \mapsto e')[e_1, .., e_n] \end{array}}{(\overline{ab}, B_p, B_{ff}, f_{in}, f_{out}, Ty, X, F) \vdash [\textbf{ffbl\_call}\, f(e_1, .., e_n)](env_A, C, \sigma, t) \longrightarrow_A [\textbf{ffbl\_call}\, f(e'_1, .., e'_n)](env_A, C, \sigma', t')}\ \text{ARC}$$

Figure 21: Architecture Reduction-of-Argument Semantics