The `p4ott` P4 Formalization

Anoud Alshnakat
Didrik Lundberg

June 23, 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 [1].

`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.

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.

bt	$::=$		base types
	$\mid$	$\mathrm{bool_t}$	
	$\mid$	$\mathrm{bit_t}$	

t	$::=$		types
	$\mid$	bt	
	$\mid$	$\mathrm{struct_t}\ t_1, \ldots, t_n$	
	$\mid$	$\mathrm{header_t}\ t_1, \ldots, t_n$	
	$\mid$	ext	

Figure 2: Types

e	$::=$		expression
	$\mid$	v	constant value
	$\mid$	**var** *varn*	variable
	$\mid$	$\{e_1, .., e_n\}$	expression list
	$\mid$	$e.x$	field access
	$\mid$	$\ominus e$	unary operation
	$\mid$	$e_1 \oplus e_2$	binary operation
	$\mid$	**concat** $e_1\ e_2$	concatenation of bit-strings
	$\mid$	$e_1[e_2 : e_3]$	bit-slice
	$\mid$	**call** $funn(e_1, .., e_n)$	function or extern call
	$\mid$	**select** $e\{v_1 : x_1; ...; v_n : x_n\}x$	select
	$\mid$	(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

$stmt$	$::=$		statement
	$\|$	$\emptyset_{\mathrm{stmt}}$	empty statement
	$\|$	$lval := e$	assignment
	$\|$	$\mathbf{if}\ e\ \mathbf{then}\ stmt_1\ \mathbf{else}\ stmt_2$	conditional
	$\|$	$\mathbf{decl}\ t\ x\ init\_opt$	declaration
	$\|$	$\{stmt\}$	block
	$\|$	$[stmt]$	block in progress
	$\|$	$\mathbf{return}\ e$	return
	$\|$	$stmt_1 ; stmt_2$	sequence
	$\|$	$\mathbf{verify}\ e\ e'$	verify
	$\|$	$\mathbf{transition}\ e$	transition
	$\|$	$\mathbf{apply}\ tbl\ e$	apply
	$\|$	$\blacksquare$	extern function

Figure 4: P4 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.

$lval$	$::=$		
	$\|$	$varn$	variable name
	$\|$	$\mathbf{null}$	null variable
	$\|$	$lval.f$	field access
	$\|$	$(lval)$	

Figure 5: P4 l-values

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.

1.4 Execution State

t	$::=$			execution status
	$\mid$	**run**		
	$\mid$	**ret** v		
	$\mid$	p		
	$\mid$	$\bot$		
σ	$::=$			execution state
	$\mid$	$(\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, t)$	M	

Figure 6: P4 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 $(\varepsilon,\ E)$, where ε is a stack of scopes γ (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.

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 $\varepsilon[0]$. 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 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:

- $\gamma :: \varepsilon$: Add a scope γ on bottom of ε (i.e. cons).
- $(i \mapsto \gamma)\ \varepsilon$: 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** . $\bot$ 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

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

$$\frac{v = \text{lookup}_{\text{v}}(\overrightarrow{\gamma}, \overrightarrow{\gamma}_L, \mathit{varn})}{\mathit{ctx}\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\textbf{var}\ \mathit{varn}) \rightsquigarrow (v, [\,])} \quad \text{E\_LOOKUP}$$

$$\frac{v = \textbf{struct}\ \{f_1 = v_1;\ \ldots;\ f_n = v_n\}(f)}{\mathit{ctx}\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\textbf{struct}\ \{f_1 = v_1;\ \ldots;\ f_n = v_n\}.f) \rightsquigarrow (v, [\,])} \quad \text{E\_S\_ACC}$$

$$\frac{v = \textbf{header}\ \mathit{boolv}\{f_1 = v_1;\ \ldots;\ f_n = v_n\}(f)}{\mathit{ctx}\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\textbf{header}\ \mathit{boolv}\{f_1 = v_1;\ \ldots;\ f_n = v_n\}.f) \rightsquigarrow (v, [\,])} \quad \text{E\_H\_ACC}$$

$$\frac{x' = \{v_1 : x_1; \ldots; v_n : x_n;\ \_ : x\}(v)}{\mathit{ctx}\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\textbf{select}\ v\{v_1 : x_1;\ \ldots;\ v_n : x_n\}x) \rightsquigarrow (x', [\,])} \quad \text{E\_SEL\_ACC}$$

Figure 7: P4 Expression Evaluation Semantics

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 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 (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 γ'. Similarly, arguments with non-out directions are simply assigned to their parameter names in γ'. Finally, γ' is put on top of the global scope γ_G in order to form the new frame of the callee 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 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.

2.2 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.

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

Figure 8: direction syntax

2.3 Statement Execution

The semantics of the statements is shown in Figures 10 and 9[2].

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

$$\frac{(\overrightarrow{\gamma}'_L, \overrightarrow{\gamma}') = \text{declare}(\overrightarrow{\gamma}_L, \overrightarrow{\gamma}, x, t)}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{decl}\, t\, x\, -)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}'_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}'}], C, \mathbf{run})} \quad \text{STMT\_DECL}$$

$$\frac{\begin{array}{l} \overrightarrow{\gamma}' = (\overrightarrow{\gamma}_L ++ \overrightarrow{\gamma})[lval \longmapsto v] \\ \gamma_G = \overrightarrow{\gamma}'[0] \\ \gamma'_G = \overrightarrow{\gamma}'[1] \\ \overrightarrow{\gamma}'' = \text{tl}(\text{tl}\, \overrightarrow{\gamma}') \end{array}}{ctx \vdash (\overrightarrow{\gamma}_L, [(lval := v)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to ([\gamma_G, \gamma'_G], [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}''}], C, \mathbf{run})} \quad \text{STMT\_ASS\_V}$$

$$\frac{}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{if}\, \top\, \mathbf{then}\, stmt_1\, \mathbf{else}\, stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, [(stmt_1)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT\_COND2}$$

$$\frac{}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{if}\, \bot\, \mathbf{then}\, stmt_1\, \mathbf{else}\, stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, [(stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT\_COND3}$$

Figure 9: P4 Statement Execution Semantics

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 ε containing the variable, then updates the mapping of the variable name x to the new value (constant v) in this scope. The 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.

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

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

$$\frac{ctx \vdash (\overrightarrow{\gamma}_L, [(stmt_1)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi} +\!\!+ [(stmt'_1)^{funn}_{\overrightarrow{\gamma}'}], C', \mathbf{run})}{ctx \vdash (\overrightarrow{\gamma}_L, [(stmt_1; stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi} +\!\!+ [(stmt'_1; stmt_2)^{funn}_{\overrightarrow{\gamma}'}], C', \mathbf{run})} \quad \textsc{stmt\_seq1}$$

$$\frac{}{ctx \vdash (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}}; stmt)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, [(stmt)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \textsc{stmt\_seq2}$$

$$\frac{\overrightarrow{\gamma}' = \overrightarrow{\gamma} + +[\gamma_\emptyset]}{ctx \vdash (\overrightarrow{\gamma}_L, [(\{stmt\})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, [([stmt])^{funn}_{\overrightarrow{\gamma}'}], C, \mathbf{run})} \quad \textsc{stmt\_block\_enter}$$

$$\frac{ctx \vdash (\overrightarrow{\gamma}_L, [(stmt)^{funn}_{\overrightarrow{\gamma}}], C, t) \to (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi} +\!\!+ [(stmt')^{funn}_{\overrightarrow{\gamma}'}], C', t')}{ctx \vdash (\overrightarrow{\gamma}_L, [([stmt])^{funn}_{\overrightarrow{\gamma}}], C, t) \to (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [([stmt'])^{funn}_{\overrightarrow{\gamma}'}], C', t')} \quad \textsc{stmt\_block\_exec}$$

$$\frac{\overrightarrow{\gamma}' = \text{rev}((\text{tl}(\text{rev}(\overrightarrow{\gamma}))))}{ctx \vdash (\overrightarrow{\gamma}_L, [([\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}}], C, t) \to (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}'}], C, t)} \quad \textsc{stmt\_block\_exit}$$

Figure 10: 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 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 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 11. 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_TRANS rules describe reduction of the **transition** state-

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

$$\frac{}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{verify}\ \top\ (\mathbf{errmsg}\, x))^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT\_VERIFY\_3}$$

$$\frac{\begin{array}{c} x' = \text{“}parseError\text{”} \\ x'' = \text{“}reject\text{”} \end{array}}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{verify}\ \bot\ (\mathbf{errmsg}\, x))^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma}_L, [(x' := (\mathbf{errmsg}\, x); \mathbf{transition}\, x'')^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT\_VERI}$$

$$\frac{\mathbf{not\_final\_state}\,(x)}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{transition}\, x)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{tra}\ x)} \quad \text{STMT\_TRANS\_1}$$

$$\frac{x = \text{“}accept\text{”}}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{transition}\, x)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{acc})} \quad \text{STMT\_TRANS\_2}$$

$$\frac{x = \text{“}reject\text{”}}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{transition}\, x)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{rej})} \quad \text{STMT\_TRANS\_3}$$

Figure 11: P4 Parser-Specific Statement Execution Semantics

ment, 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)

$\boxed{ctx_P \vdash \sigma \longrightarrow_p \sigma'}$ parser block semantics

$$\frac{\begin{array}{l} \mathbf{parser\_not\_finished}\ \overrightarrow{\Phi} \\ (X, F, \mathbf{empty}) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \rightarrow \sigma' \end{array}}{(X, F, P) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_p \sigma'} \quad \text{PARS\_STMT}$$

$$\frac{stmt'' = P(x)}{(X, F, P) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{tra}\ x) \longrightarrow_p (\overrightarrow{\gamma}_L, [(stmt'')^{x}_{[\gamma_\emptyset]}], C, \mathbf{run})} \quad \text{PARS\_STATE}$$

$$\frac{x = \text{“}reject\text{”}}{(X, F, P) \vdash (\overrightarrow{\gamma}_L, [(\emptyset_{\text{stmt}})^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \longrightarrow_p (\overrightarrow{\gamma}_L, [(\mathbf{transition}\, x)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{PARS\_EMPTY}$$

Figure 12: Parser Block-Level Semantics

The parser state machine semantics is shown in Figure 12.

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 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.

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(\mathbf{apply}\ tbl\ e)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} \mathbin{+\!\!+} [(\mathbf{apply}\ tbl\ e')^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})}\quad \text{STMT\_APPLY\_TABLE\_E}$$

$$\frac{\begin{array}{l}\mathrm{Tb}\, tbl\ =\ (e'\,, mk) \\ \mathrm{ctrl}\,(tbl,\ v,\ mk\,) = (f, (v_1,\ ..\,, v_n))\end{array}}{(X, F, Tb) \vdash (\overrightarrow{\gamma}_L, [(\mathbf{apply}\ tbl\ v)^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma}_L, [(\mathbf{null} := (\mathbf{call}\ f\,(v_1,\ ..\,, v_n)))^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})}\quad \text{STMT\_APPLY\_TABL}$$

Figure 13: Match Action Execution Semantics

$\boxed{ctx_C \vdash \sigma \longrightarrow_c \sigma'}$ control block semantics

$$\frac{(X, F, Tb) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \to (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')}{(X, F, Tb) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_c (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')}\quad \text{CTRL\_STMT}$$

Figure 14: Control Block-Level Semantics

The statement semantics unique to the control block is shown in Figure 13. 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 14. 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 at the top of Figure 15, consists of multiple components: starting from the left, an *architectural context* ctx_A (which contains everything not changed by reduction steps) on the left-hand side of the turnstile, which contains the following:

1. The *architectural block list* $\overline{ab}$: an architectural block represents a stage of packet processing. There are four fundamental stages of packet processing in a P4-compatible architecture. First, there are the input (**inp**) and output (**out**) stages: these stages just perform translation between the architectural packet format and the generic I/O format (consisting of a list of ports, each with pending packets to be arbitrated/sent off)[3]. Second, there are the programmable blocks (parser blocks and control blocks, which are described in more detail elsewhere in this document) and the fixed-function blocks. Fixed-function blocks, like their name implies, perform the parts of packet processing in the P4-programmable network element that are actually not P4-programmable.

DL: This currently just a list, not a list of lists. Fix the Ott+HOL4, then env_A below to agree with the description here.

2. The *programmable block map* B_p, which is a partial map between names of programmable block names (strings) and the all necessary items that model the block in question. For parser blocks, this is the list of directed parameters, a statement representing the declarations and instantiations done at the block-global level, and the parser state map P between parser state names and their bodies (statements)[4]. For control blocks, this is the list of directed parameters, a statement representing the declarations and instantiations done at the block-global level, the body of the control block (found in the topmost apply statement)[5] and a table map Tb between names of tables and tuples of expressions and matching kinds.

3. The *fixed-function block map* B_{ff}, which is a partial map between names of fixed-function blocks to the implementation of the fixed function itself, which is a partial function that maps the architectural scope to an updated architectural scope.

DL: Why not include control plane configuration?

4. The *input function* f_{in} and the *output function* f_{out}, which are both partial functions from an IO list and the architectural scope to an updated IO list and architectural scope. They are used in the input and output stages.

5. The *programmable block copy-in function* $copyin_{pbl}$ and the *programmable block copy-out function* $copyout_{pbl}$: $copyin_{pbl}$ is a partial function from a list of directed arguments, a list of expressions and the architectural scope to a scope (the block-global scope of the programmable block). $copyout_{pbl}$ is a partial function from the global scopes list, a list of directed arguments and the architectural scope to an updated architectural scope.

[3]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

[4]Note that `p4ott` represents all parser state bodies as encased in a block

[5]Note that `p4ott` represents the content of the apply as encased in a block

6. The *extern object map* X, which is a partial map from extern object names to tuples of extern function maps (a map holding the extern functions of the object: tuples of directed parameter lists, function bodies, and the extern implementation), and an optional constructor (holding a tuple same as that of the extern function map).

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

The left-hand and right-hand sides of the reduction both have the same elements.

1. The *architectural environment* env_A has five components:
   (a) The *architectural block index*: This is an index which informs us of which architectural block in $\overline{ab}$ is currently being reduced.
   (b) The *input list*, which is a list of incoming packets (represented as tuples of lists of Booleans and numbers signifying input ports)
   (c) The *output list*, which is the same as the input list, but used for output.
   (d) The *architectural scope* γ_A, which is of polymorphic type, and is used for storing things in-between the programmable blocks, as well as things that are not accessible directly by P4 code.

2. The global scope list $\bar{\gamma}_L$ containing the top-level global scope with constants common to all programmable blocks as well as the block-global scope containing variables declared at the start of progrmamable blocks.

3. The *architecture-level frame list*: this contains either a regular frame list, or a special empty architecture-level frame list ($[\,]_A$).

   DL: Three solutions to the problem of recognizing in-progress blocks: (1) add a Boolean in-progress flag to the state, (2) use an architecture-level frame list with this special symbol (2) use a regular empty frame list and special checks in the EXEC rules. Solution 2 is probably best, since it takes up the least additional space in the rules (no additional premise or tuple element)

DL: This could be assigned as symbol instead of being called "*arch_frame_list*" in the judgment

4. The control-plane configuration C, a partial function from strings, values and match kinds to tuples of strings and expression lists.

5. The status, which (other than the functionality on other levels of the semantics) informs us of whether a parser state machine is finished, or the apply of a control block is finished.

The rules of the architecture-level semantics are the following:

1. ARCH_IN: The first antecedent requires that the pending architecture block is **inp**, and then f_{in} updates the input list and the architectural scope.

2. ARCH_OUT: Similar to the above, but the pending architecture block must be **out**, and then the output list and the architectural scope are updated by f_{out}.

3. ARCH_PARSER_INIT: when the pending architecture block is a parser block. Then, a new block-global scope γ' is created and initialised with the arguments of the parser block as well as *parseError*, which is initially set to *NoError*. The result is appended to the top-level global scope. The architecture frame list is set to a single frame where the statement is the declarations of the parser block *stmt* followed by the body of the *start* block *stmt'*, the scopes stack is the empty list and the current function name is the name of the parser block.

4. ARCH_CONTROL_INIT is similar, but simpler: here, *parseError* is not set, and the body of the *start* block is replaced with the body of the apply *stmt'*.

5. ARCH_FFBL handles the fixed-function block: when the pending architecture block is a fixed-function block with name x, the implementation *ff* of the fixed-function is looked-up in B_{ff} using x, after which it is used to update the architectural scope. Also, the architecture block index is incremented.

6. ARCH_PARSER_EXEC describes the reduction of the parser block: the first two premises are there to ensure the current block is a parser block. It is also required that the status t not be *pars_fin*, i.e., the parser has not yet finished. Then, the global block list, the frame list, the control plane configuration and the status are updated by a parser reduction step $\longrightarrow_p$.

7. ARCH_CONTROL_EXEC is similar to the above, but ensures the current block is a control block, after which the control reduction step $\longrightarrow_c$ is used to update the architectural state.

8. ARCH_PARSER_RET rule describes the final step of parser block reduction. The first two premises ensure the current block is a parser block and provide the directed parameters for $copyout_{pbl}$ to update the architectural scope. Furthermore, the architecture block list index is incremented by 1 and the block-global scope is dropped from the global scopes list.

9. ARCH_CONTROL_RET is analoguous to the above in every aspect, but for control blocks.

$\boxed{ctx_A \vdash (env_A, \overrightarrow{\gamma}_L, arch\_frame\_list, C, t) \longrightarrow_A (env_A{}', \overrightarrow{\gamma}'_L, arch\_frame\_list', C', t')}$

architecture-level semantics

$$\frac{\begin{array}{l}\mathbf{inp} = \overline{ab}[i]\\ (\overline{io}'', \gamma_A{}') = in_A(\overline{io}, \gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i+1, \overline{io}'', \overline{io}', \gamma_A{}'), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run})} \quad \textsc{arch\_in}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{parser}\, ((x_1, d_1), .., (x_n, d_n))stmt\, P = B_p(f)\\ x = \text{``}start\text{''}\\ stmt' = P(x)\\ \gamma' = in_p((x_1, .., x_n), [d_1, .., d_n], [e_1, .., e_n], \gamma_A)\\ \gamma''_G = \overrightarrow{\gamma}_L[0]\\ \overrightarrow{\gamma}'_L = [\gamma''_G] + +[\gamma']\\ v = NoError\\ x' = parseError\\ \overrightarrow{\gamma}''_L = \text{initialise}(\overrightarrow{\gamma}'_L, x', v)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}''_L, [(stmt; stmt')^f_{[\,]}], C, \mathbf{run})} \quad \textsc{arch\_parser\_init}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{control}\, ((x_1, d_1), .., (x_n, d_n))stmt\, stmt'\, Tb = B_p(f)\\ \gamma' = in_p((x_1, .., x_n), [d_1, .., d_n], [e_1, .., e_n], \gamma_A)\\ \gamma''_G = \overrightarrow{\gamma}_L[0]\\ \overrightarrow{\gamma}'_L = [\gamma''_G] + +[\gamma']\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}'_L, [(stmt; stmt')^f_{[\,]}], C, \mathbf{run})} \quad \textsc{arch\_control\_init}$$

$$\frac{\begin{array}{l}\mathbf{ffbl}\, x = \overline{ab}[i]\\ ff = B_{ff}(x)\\ \gamma_A{}' = ff(\gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i+1, \overline{io}, \overline{io}', \gamma_A{}'), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run})} \quad \textsc{arch\_ffbl}$$

$$\frac{\begin{array}{l}\mathbf{out} = \overline{ab}[i]\\ (\overline{io}'', \gamma_A{}') = out_A(\overline{io}', \gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((0, \overline{io}, \overline{io}'', \gamma_A{}'), \overrightarrow{\gamma}_L, [\,]_A, C, \mathbf{run})} \quad \textsc{arch\_out}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, x(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{parser}\, ((x_1, d_1), .., (x_n, d_n))stmt\, P = B_p(x)\\ t \neq pars\_fin\\ (X, F, P) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, t) \longrightarrow_p (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, t) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')} \quad \textsc{arch\_parser\_exec}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, x(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{control}\, ((x_1, d_1), .., (x_n, d_n))stmt\, stmt'\, Tb = B_p(x)\\ (X, F, Tb) \vdash (\overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_c (\overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}_L, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma}'_L, \overrightarrow{\Phi}', C', t')} \quad \textsc{arch\_control\_exec}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{parser}\, ((x_1, d_1), .., (x_n, d_n))stmt\, P = B_p(f)\\ \gamma_A{}' = out_p([\gamma_G, \gamma'_G], \gamma_A, [d_1, .., d_n], (x_1, .., x_n))\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), [\gamma_G, \gamma'_G], \overrightarrow{\Phi}, C, p_{\text{fin}}) \longrightarrow_A ((i+1, \overline{io}, \overline{io}', \gamma_A{}'), [\gamma_G], [\,]_A, C, \mathbf{run})} \quad \textsc{arch\_parser\_ret}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i]\\ \mathbf{control}\, ((x_1, d_1), .., (x_n, d_n))stmt\, stmt'\, Tb = B_p(f)\\ \gamma_A{}' = out_p([\gamma_G, \gamma'_G], \gamma_A, [d_1, .., d_n], (x_1, .., x_n))\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), [\gamma_G, \gamma'_G], \overrightarrow{\Phi}, C, \mathbf{ret}\, v) \longrightarrow_A ((i+1, \overline{io}, \overline{io}', \gamma_A{}'), [\gamma_G], [\,]_A, C, \mathbf{run})} \quad \textsc{arch\_control\_ret}$$

Figure 15: Architecture-Level Semantics

A Concrete Syntax of Operations

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

Figure 16: P4 Unary Operations

The unary expressions included are shown in Figure 16. 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).

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

Figure 17: P4 Binary Operations

The binary expressions included are shown in Figure 16. 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 18. The statement semantics are shown in Figure 19.

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).

References

[1] Ryan Doenges et al. “Petr4: formal foundations for p4 data planes”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–32.

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

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\textbf{select}\ e\{v_1 : x_1; \ldots; v_n : x_n\}x) \rightsquigarrow (\textbf{select}\ e'\{v_1 : x_1; \ldots; v_n : x_n\}x, \overrightarrow{\Phi})} \quad \text{E\_SEL\_ARG}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (\ominus e) \rightsquigarrow (\ominus e', \overrightarrow{\Phi})} \quad \text{E\_UNOP\_ARG}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e'', \overrightarrow{\Phi})}{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e \oplus e') \rightsquigarrow (e'' \oplus e', \overrightarrow{\Phi})} \quad \text{E\_BINOP\_ARG1}$$

$$\frac{\begin{array}{c}\text{is\_short\_circuit}(\oplus) \\ ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})\end{array}}{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (v \oplus e) \rightsquigarrow (v \oplus e', \overrightarrow{\Phi})} \quad \text{E\_BINOP\_ARG2}$$

Figure 18: Expression Reduction-of-Argument Semantics

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

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(\textbf{return}\ e)^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [(\textbf{return}\ e')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run})} \quad \text{STMT\_RET\_E}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(lval := e)^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [(lval := e')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run})} \quad \text{STMT\_ASS\_E}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(\textbf{if}\ e\ \textbf{then}\ stmt_1\ \textbf{else}\ stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [(\textbf{if}\ e'\ \textbf{then}\ stmt_1\ \textbf{else}\ stmt_2)^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run})} \quad \text{STMT\_CO}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e'', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(\textbf{verify}\ e\ e')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [(\textbf{verify}\ e''\ e')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run})} \quad \text{STMT\_VERIFY\_E1}$$

$$\frac{ctx\ \overrightarrow{\gamma}_L\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})}{ctx \vdash (\overrightarrow{\gamma}_L, [(\textbf{verify}\ b\ e)^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma}_L, \overrightarrow{\Phi} +\!\!+ [(\textbf{verify}\ b\ e')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run})} \quad \text{STMT\_VERIFY\_E2}$$

Figure 19: Statement Reduction-of-Argument Semantics