The p4ott P4 Formalization

Anoud Alshnakat
Didrik Lundberg

August 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), tbl (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
		bool_t	
		bit_t	

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

Figure 2: Types

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

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
	|	$\emptyset_{\text{stmt}}$	empty statement
	|	$lval := e$	assignment
	|	**if** e **then** $stmt_1$ **else** $stmt_2$	conditional
	|	$\{\overrightarrow{decl}\ stmt\}$	block
	|	**return** e	return
	|	$stmt_1; stmt_2$	sequence
	|	**verify** $e\ e'$	verify
	|	**transition** e	transition
	|	**apply** $tbl(e_1, .., e_n)$	apply
	|	■	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. The **verify** statement (modeled as a statement and not as 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 the 'reject" parser state with the 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 *tbl* (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
	|	**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
	|	**run**	
	|	**ret** v	
	|	**tra** x	transition to parser state
	|	$\bot$	
s	::=		execution state
	|	$(\overrightarrow{\gamma_G}, \overrightarrow{\Phi}, C, t)$ 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 global scope list $\overrightarrow{\gamma}_G$, a frame list $\overrightarrow{\Phi}$, Control table C, and the state status t.

More formally, a scope $\gamma : X \hookrightarrow V * (X \cup \{\bot\})$ 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 **run** 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

tra x signifies transition to a new parser state - a final state (p_{fin}) in the case of "accept" or 'reject", or otherwise a state defined by the programmer. $\perp$ 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.

1.5 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 ::= $\downarrow \mid \uparrow \mid \updownarrow \mid \circ$
Due to the calling conventions, the scope has the type; $\gamma : X \hookrightarrow V * (X \cup \{\perp\})$.
The list of scopes is identified with an arrow following $\overrightarrow{\gamma}$. The local $\overrightarrow{\gamma}$ is meant to be a local list of scopes to the frame. The global $\overrightarrow{\gamma}_G$ is a list with a length of 2. The first index determines the external architecture scope (can not be defined by a P4 program), while the second index determines the architecture local variables (defined in the architecture level). Opeations on list of scopes:

1. Operation $\gamma[x \longmapsto v]$ to update a variable name x with value v in the scope γ.
2. Operation $\overrightarrow{\gamma}[x \longmapsto v]$ to update a variable name x with value v in the most recent scope that contains the variable name x.
3. Operation $lookup_v(\overrightarrow{\gamma}, \overrightarrow{\gamma}_G, x)$ to return the value v of the tuple $(y, x \cup \perp)$, which is whatever variable x is mapped to in the most recent scope it is defined in after concatenating $\overrightarrow{\gamma}_G +\!\!+ \overrightarrow{\gamma}$.

2 Semantics

2.1 Expressions

Expression semantics and reductions are layed out in this section. Overall, the reductions can never alter or have any side effects on the state. The only thing it can perform is a reduction on the expressions -standard small step structural semantics- and also produce a new frame -which occurs only in the function call reduction-.

variable lookup

In the E_LOOKUP rule, the lookup function ensures that the variable name x is evaluated in the uppermost (i.e. most recently scope γ that x is declared or instantiated in). The evaluation will occur in the global list of scopes $\overrightarrow{\gamma}_G$ with the local $\overrightarrow{\gamma}$ being the most recent scopes. 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.

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

function call

Note that function calls also include actions as well as extern function calls.

The E_CALL_ARGS is used whenever there is a function call expression with unreduced (in and none directioned arguments). Each function's argument will be checked against the function's direction in the same position. If the direction is in or none, then it will be reduced until it becomes a constant, otherwise if the argument has the direction out or inout then do not reduce it.

The E_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). The function call reduction will produce a placeholder we call **var star** and a new frame. The new frame will be later added on top of the state's frames in the statements reduction rules. The way that this new frame is constructed is as follows: The function name $funn$ will be the same as the call in the expression. The function name $funn$ will be looked up in both the function maps and extern maps to retrieve the function body $stmt$ and the signature of the parameters represented as a list of tupels: variable names and their directions $[(x_1, d_1), \ldots, (x_n, d_n)]$. Each argument will be checked against the boolean formula to ensure that the arguments were reduced properly: (if d_i in-none then the expression in the same position should be a constant, otherwise a variable name). The new frame's scope γ' is a new fresh empty scope that contains the copied in arguments using the function **copyin**.

headers and structs

The struct can be an expression, thus we need reductions for it. The rule E_ESTRUCT rule will reduce the expression feilds one at a time from left to right. One all the expressions become constants then we can transform the expression struct to a value struct via E_ESTRUCT_TO_V which will be used mostly in header access. Same extact operations are applied on the headers.

$$\frac{\begin{array}{l}[(x_1, d_1), .., (x_n, d_n)] = \textbf{lookup\_funn\_sig}\,(funn, F_g, F_b, X)\\ i = \min\{j.\ [d_1, .., d_n][j] \in \{\circ, \downarrow\} \wedge \neg(\text{is\_const}\,[e_1, .., e_n][j])\}\\ e = [e_1, .., e_n][i]\\ (X, F_g, F_b, P, Tb)\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})\\ [e'_1, .., e'_n] = (i \mapsto e')[e_1, .., e_n]\end{array}}{(X, F_g, F_b, P, Tb)\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{call}\,funn(e_1, .., e_n)) \rightsquigarrow (\textbf{call}\,funn(e'_1, .., e'_n), \overrightarrow{\Phi})}\ \textsc{e\_call\_args}$$

$$\frac{\begin{array}{l}(stmt, [(x_1, d_1), .., (x_n, d_n)]) = \textbf{lookup\_funn\_sig\_body}\,(funn, F_g, F_b, X)\\ \forall i \leqslant n.\ ((d_i \in \{\circ, \downarrow\} \implies \text{is\_const}\,e_i) \wedge (d_i \in \{\updownarrow, \uparrow\} \implies \text{is\_var}\,e_i))\\ \gamma' = \text{copyin}([x_1, .., x_n], [e_1, .., e_n], [d_1, .., d_n], , , \overrightarrow{\gamma})\end{array}}{(X, F_g, F_b, P, Tb)\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{call}\,funn(e_1, .., e_n)) \rightsquigarrow (\textbf{var}\,(\textbf{star}, funn), [([stmt])^{funn}_{[\gamma']}])}\ \textsc{e\_call\_newframe}$$

$$\frac{\begin{array}{l}i = \min\{j.\ \neg(\text{is\_const}\,[e_1, \ldots, e_n][j])\}\\ e = [e_1, \ldots, e_n][i]\\ ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})\\ [e'_1, \ldots, e'_n] = (i \mapsto e')[e_1, \ldots, e_n]\end{array}}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{eStruct}\,\{f_1 = e_1; \ldots; f_n = e_n\}) \rightsquigarrow (\textbf{eStruct}\,\{f_1 = e'_1; \ldots; f_n = e'_n\}, \overrightarrow{\Phi})}\ \textsc{e\_eStruct}$$

$$\frac{\begin{array}{l}\text{is\_consts}\ e_1, \ldots, e_n\\ v_1, \ldots, v_n = \text{vl\_of\_el}\,(e_1, \ldots, e_n)\end{array}}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{eStruct}\,\{f_1 = e_1; \ldots; f_n = e_n\}) \rightsquigarrow (\textbf{struct}\,\{f_1 = v_1; \ldots; f_n = v_n\}, [\,])}\ \textsc{e\_eStruct\_to\_v}$$

$$\frac{\begin{array}{l}i = \min\{j.\ \neg(\text{is\_const}\,[e_1, \ldots, e_n][j])\}\\ e = [e_1, \ldots, e_n][i]\\ ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi})\\ [e'_1, \ldots, e'_n] = (i \mapsto e')[e_1, \ldots, e_n]\end{array}}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{eHeader}\,boolv\{f_1 = e_1; \ldots; f_n = e_n\}) \rightsquigarrow (\textbf{eHeader}\,boolv\{f_1 = e'_1; \ldots; f_n = e'_n\}, \overrightarrow{\Phi})}\ \textsc{e\_eHeader}$$

$$\frac{\begin{array}{l}\text{is\_consts}\ e_1, \ldots, e_n\\ v_1, \ldots, v_n = \text{vl\_of\_el}\,(e_1, \ldots, e_n)\end{array}}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{eHeader}\,boolv\{f_1 = e_1; \ldots; f_n = e_n\}) \rightsquigarrow (\textbf{header}\,boolv\{f_1 = v_1; \ldots; f_n = v_n\}, [\,])}\ \textsc{e\_eHeader\_to\_v}$$

access headers and structs

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.

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

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

select label

The E_SEL_ACC rule is used to match the given value v against the label-value list, in the case a match exists. If the match doesn't exist, then return the default label x.

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

bit slicing

This reduction gives a bitvector $bitv'''$ that is reduced from the slicing operation. It extracts a contiguous list from the opriginal $bitv$ from the lsb $bitv$" till the msb $bitv$'.

$$\frac{bitv''' = bitv[bitv' : bitv'']}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (bitv[bitv' : bitv'']) \rightsquigarrow (bitv''', [\,])} \quad \text{E\_SLICE\_V}$$

concatenation

The reduction produces one bit string $bitv''$ that is the result of concatenating two bit strings $bitv$ and $bitv'$.

$$\frac{bitv'' = bitv +\!\!+ bitv'}{ctx\,\overrightarrow{\gamma_G}\,\overrightarrow{\gamma} \vdash (\textbf{concat}\,bitv\;bitv') \rightsquigarrow (bitv'', [\,])} \quad \text{E\_CONCAT\_V}$$

unary and binary

Unary and binary operations are basically standard operations on bit vectors, therefore are removed from this section.

2.2 Statement (single frame) semantics

The semantics of the statements are presented in this section. [2].

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

assignment

The assignment can assign to *lval*s (shown in Figure 5), they can be either variable identified by their names, a null variable (used to model method calls) or struct fields, which are identified by the struct and field names, similar to the field access expression. Whenever an expression is assigned to an *lval*, that expression shall be reduced until it becomes a constant value. Then the appropriate scope in the current frame of execution will be updated with the mapping $x \mapsto v$. With appropriate we mean that the topmost (most recent) scope that the *lval* is declared in. The reduction results in the empty statement and an updated local or global scope lists. Note that this doesn't apply on **null** since it is used for method and action calls with no return values in mind.

$$\frac{\begin{array}{l}\overrightarrow{\gamma}' = (\overrightarrow{\gamma_G} + + \overrightarrow{\gamma})[lval \longmapsto v] \\ \overrightarrow{\gamma_G}' = \text{take}(2, \overrightarrow{\gamma}') \\ \overrightarrow{\gamma}'' = \text{drop}(2, \overrightarrow{\gamma}')\end{array}}{ctx \vdash (\overrightarrow{\gamma_G}, [([lval := v])^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma_G}', [([\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}''}], C, \textbf{run})} \text{ STMT\_ASS\_V}$$

if then else

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

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

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

block

The {} brackets indicate a block, while the [ ] brackets indicate a block in progress or being executed.

Once a block statement is encountered the STMT_BLOCK_ENTER reduction will be used, it entails the $\overrightarrow{decl}$ being transformed and replicated into a scope that will be appended to the local $\overrightarrow{\gamma}$ of the frame, 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 scope list $\overrightarrow{\gamma}$. The block will be exited and reduced to an empty statement $\emptyset_{\text{stmt}}$.

apply

The STMT_APPLY_V describes the apply table statement. Each apply statement has a table name *tbl* with the a list of key expressions $e_1, \ldots, e_n$ to match on. This list have been previously reduced

$$\frac{\begin{array}{l}\gamma = \text{replicate}(\overrightarrow{decl}) \\ \overrightarrow{\gamma}' = \overrightarrow{\gamma} \; +\!+ [\gamma]\end{array}}{ctx \vdash (\overrightarrow{\gamma_G}, [([\{\overrightarrow{decl}\; stmt\}])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma_G}, [(stmt :: [\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}'}], C, \mathbf{run})} \quad \text{STMT\_BLOCK\_ENTER}$$

$$\frac{\begin{array}{l}\mathbf{not\_empty}\ stmt \\ (X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [([stmt])^{funn}_{\overrightarrow{\gamma}}], C, t) \rightarrow (\overrightarrow{\gamma_G}'', \overrightarrow{\Phi}' +\!+ [(\overrightarrow{stmt}')^{funn}_{\overrightarrow{\gamma}'}], C', t')\end{array}}{(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [(stmt :: \overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}], C, t) \rightarrow (\overrightarrow{\gamma_G}', \overrightarrow{\Phi}' +\!+ [(\overrightarrow{stmt}' \; +\!+ \overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}'}], C', t')} \quad \text{STMT\_BLOCK\_EXEC}$$

$$\frac{\overrightarrow{\gamma}' = \text{rev}((\text{tl}(\text{rev}(\overrightarrow{\gamma}))))}{ctx \vdash (\overrightarrow{\gamma_G}, [(\emptyset_{\text{stmt}} :: \overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}], C, t) \rightarrow (\overrightarrow{\gamma_G}, [(\overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}'}], C, t)} \quad \text{STMT\_BLOCK\_EXIT}$$

to constants one at a time in the rule STMT_APPLY_E. The Tb will return a list of match kind $mk_1, ..., mk_n$. The ctrl is a function that maps table names tbl, expressions, and the match kinds to action that should be implemented as a method call, with a list of the arguments $v_1, ..., v_m$.

$$\frac{\begin{array}{l}i = \min\{j. \neg(\text{is\_const}\,[e_1, .., e_n][j])\} \\ e = [e_1, .., e_n][i] \\ ctx\ \overrightarrow{\gamma_G}\ \overrightarrow{\gamma} \vdash (e) \rightsquigarrow (e', \overrightarrow{\Phi}) \\ [e_1', .., e_n'] = (i \mapsto e')[e_1, .., e_n]\end{array}}{ctx \vdash (\overrightarrow{\gamma_G}, [([\mathbf{apply}\; tbl(e_1, .., e_n)])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma_G}, \overrightarrow{\Phi} +\!+ [([\mathbf{apply}\; tbl(e_1', .., e_n')])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT\_APPLY\_TABL}$$

$$\frac{\begin{array}{l}\text{is\_consts}\ e_1, .., e_n \\ \text{Tb}\, tbl\ =\ ([mk_1, .., mk_n]) \\ \text{C}\,(tbl, (e_1, .., e_n), ([mk_1, .., mk_n]))\ =\ (f, (v_1, .., v_m))\end{array}}{(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [([\mathbf{apply}\; tbl(e_1, .., e_n)])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma_G}, [([\mathbf{null} := (\mathbf{call}\; f(v_1, .., v_m))])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \quad \text{STMT}$$

return

Once return's expression is reduced to a constant value, the status is changed to a **ret** v, and the statement becomes $\emptyset_{\text{stmt}}$. The rest of the return operatoin will be handeled in the sequence and composition rules.

$$\frac{}{ctx \vdash (\overrightarrow{\gamma_G}, [([\mathbf{return}\; v])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma_G}, [([\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{ret}\; v)} \quad \text{STMT\_RET\_V}$$

sequential statements

The sequential statements rules STMT_SEQ1 and STMT_SEQ2 are standard. The status t must be **run** to start with. The part that is different is that if any new frame being created by any function call, it will be added on top of the frame list.

Otherwise, the STMT_SEQ3 rule is used whenever the status is **ret** , or **tra** to indicate either a a transition to a parser state or a return from a function call. The rest of the sequential composition statements are not considered, this will give flexibility to the return or transition statements in the body rather than only at the end. STMT_SEQ1 and STMT_SEQ3 are able to change the tables that are populated by the control plane.

$$\frac{ctx \vdash (\overrightarrow{\gamma_G}, [([stmt_1])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}', \overrightarrow{\Phi} +\!\!+ [(\overrightarrow{stmt}' + + [stmt_1'])^{funn}_{\overrightarrow{\gamma}'}], C', \mathbf{run})}{ctx \vdash (\overrightarrow{\gamma_G}, [([stmt_1; stmt_2])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}', \overrightarrow{\Phi} +\!\!+ [(\overrightarrow{stmt}' + + [stmt_1'; stmt_2])^{funn}_{\overrightarrow{\gamma}'}], C', \mathbf{run})} \text{ STMT\_SEQ1}$$

$$\frac{}{ctx \vdash (\overrightarrow{\gamma_G}, [([\emptyset_{\text{stmt}}; stmt])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}, [([stmt])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run})} \text{ STMT\_SEQ2}$$

$$\frac{\begin{array}{l} ctx \vdash (\overrightarrow{\gamma_G}, [([stmt_1])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}', [([stmt_1'])^{funn}_{\overrightarrow{\gamma}'}], C', t) \\ t \neq \mathbf{run} \end{array}}{ctx \vdash (\overrightarrow{\gamma_G}, [([stmt_1; stmt_2])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}', [([stmt_1'])^{funn}_{\overrightarrow{\gamma}'}], C', t)} \text{ STMT\_SEQ3}$$

extern

The symbol ■ represents a statement capturing the semantics of an extern function other than copy-in copy-out and return behaviour. ■ is able to modify the global scope list $\bar{\gamma}_G$, the local scope list $\overrightarrow{\gamma}$ and the control plane configuration C. The exact behaviour is determined by looking up the entry of the name *funn* associated with the current frame in the extern function map X: since ■ is used in extern function call bodies, *funn* is the name of the extern function.

Note that calling an extern function works the same as calling any other function, as described in the function call subsection of Section 2.1. However, by convention the body of the extern function consists only of ■, possibly followed by a **return** statement if the extern function in question has any return value.

$$\frac{\begin{array}{l} ext\_fun = \text{lookup\_ext\_fun}(funn, X) \\ (\overrightarrow{\gamma_G}', \overrightarrow{\gamma}', C') = ext\_fun\,(\overrightarrow{\gamma_G}, \overrightarrow{\gamma}, C) \end{array}}{(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [([\blacksquare])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \to (\overrightarrow{\gamma_G}', [([\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}'}], C', \mathbf{run})} \text{ STMT\_EXT}$$

transition

The transition statement **transition** x, whose semantics is captured by the STMT_TRANS rule, makes a change to the state's status t based on the state name x.

$$\frac{}{ctx \vdash (\overrightarrow{\gamma_G}, [([\mathbf{transition}\, x])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{run}) \rightarrow (\overrightarrow{\gamma_G}, [([\emptyset_{\text{stmt}}])^{funn}_{\overrightarrow{\gamma}}], C, \mathbf{tra}\ x)} \quad \text{STMT\_TRANS}$$

verify

The **verify** $e\, e'$ statement is used to check the whether the boolean expression e holds; if it does, then nothing happens, as in the STMT_VERIFY_3 rule. Otherwise, it assigns the error in e' to "*parseError*" and reduces the statement to a **transition** statement to the state 'reject".

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

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

2.3 list of frames semantics

The previous semantics operate on a single frame for a single statement, however whenever a function being called, then it means that the state will contain two frames. The top frame that will be executed as can be seen in STMT_COMP1 reduction, where the previous frame Φ is not empty (otherwise, we can use the previous rules above). The rule STMT_COMP2 is used specifically whenever there is a return occuring in a frame in that is being reduced. Here the caller frame's scopelist $\overrightarrow{\gamma}'$ or the global scope list $\overrightarrow{\gamma}_G$ should initilize the **var star** with the return that can be found in the state's status **ret** v. the proper scope will be updated based on the topmost scope ever that can be found after concatenating $\overrightarrow{\gamma}_G +\!\!+ \overrightarrow{\gamma}$ to result the scope list $\overrightarrow{\gamma}''$. The copy out function takes the callee function signature, the "possibly updated" global scope list (index 0 and 1 from $\overrightarrow{\gamma}''$) the top most scope in the $\overrightarrow{\gamma}''$ and the callee's current scope list $\overrightarrow{\gamma}$. It returns a new updated scopes list for both the global and caller's frame. In the copyout function, each parameter with (out or inout directions) should be copied out to the variable name it is mapping to. i.e. the mapping of a single scope looks as $x \mapsto (v, y)$ so for each variable name x that is part of the current argument list in the signature we will check the direction of it, if the direction is (none or in) then we do not copy them out. However, if the direction is (out or inout) we will look into the mapping of it, and retrive a tuple (v, y) which is the value and the oprginal argument of the function to be copied out in the poroper scope (top most one it is defined in).

We define scopes_to_pass $(funn, F_g, F_b, \overrightarrow{\gamma_G})$ as :

$$\frac{\begin{array}{l}\overrightarrow{\gamma_G}' = \text{scopes\_to\_pass}(funn, F_g, F_b, \overrightarrow{\gamma_G}) \\ \overrightarrow{\Phi}'' \neq [] \Rightarrow t' \neq \textbf{ret v} \\ (X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}', [(\overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}], C, t) \rightarrow (\overrightarrow{\gamma_G}'', \overrightarrow{\Phi}', C', t') \\ \overrightarrow{\gamma_G}''' = \text{scopes\_to\_retrieve}(funn, F_g, F_b, \overrightarrow{\gamma_G}, \overrightarrow{\gamma_G}'')\end{array}}{(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [(\overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}] +\!\!+ \overrightarrow{\Phi}'', C, t) \longrightarrow_\Phi (\overrightarrow{\gamma_G}''', \overrightarrow{\Phi}' +\!\!+ \overrightarrow{\Phi}'', C', t')} \text{ FRAMES\_COMP1}$$

$$\frac{\begin{array}{l}(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [(\overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}], C, \textbf{run}) \rightarrow (\overrightarrow{\gamma_G}, [(\overrightarrow{stmt}'')^{funn}_{\overrightarrow{\gamma}}], C, \textbf{ret } v) \\ \overrightarrow{\gamma}'' = (\overrightarrow{\gamma_G} + + \overrightarrow{\gamma}')[(\textbf{star}, funn) \longmapsto v] \\ \overrightarrow{\gamma_G}' = \text{take}(2, \overrightarrow{\gamma}'') \\ \overrightarrow{\gamma}''' = \text{drop}(2, \overrightarrow{\gamma}'') \\ (stmt''', [(x_1, d_1), .., (x_n, d_n)]) = \textbf{lookup\_funn\_sig\_body}\,(funn, F_g, F_b, X) \\ \overrightarrow{\gamma_G}'' = \text{scopes\_to\_retrieve}(funn, F_g, F_b, \overrightarrow{\gamma_G}, \overrightarrow{\gamma_G}') \\ (\overrightarrow{\gamma_G}''', \overrightarrow{\gamma}'''') = \text{copyout}\,([x1, .., xn], [d1, .., dn], \overrightarrow{\gamma_G}', \overrightarrow{\gamma}''', \overrightarrow{\gamma})\end{array}}{(X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, [(\overrightarrow{stmt})^{funn}_{\overrightarrow{\gamma}}] +\!\!+ [(\overrightarrow{stmt'})^{funn'}_{\overrightarrow{\gamma}'}] +\!\!+ \overrightarrow{\Phi}, C, \textbf{run}) \longrightarrow_\Phi (\overrightarrow{\gamma_G}''', [(\overrightarrow{stmt'})^{funn'}_{\overrightarrow{\gamma}''''}] +\!\!+ \overrightarrow{\Phi}, C, \textbf{run})} \text{ FRAMES\_COM}$$

$$\overrightarrow{\gamma_G}' = \begin{cases} \text{if } funn \in dom(F_g) & \text{, then } [\overrightarrow{\gamma_G}[0]; \emptyset_\gamma] \\ \text{otherwise} & \text{, then } \overrightarrow{\gamma_G} \end{cases}$$

Where $\overrightarrow{\gamma_G}$ is a list of two elements, index 0 is the global scope, and index 1 is the programmable block global scope, it is fetched from the frame reduction rule. $\overrightarrow{\gamma_G}'$ is the scope that will be used to implement one frame at a time (statement reduction) in comp2.

We define scopes_to_retrieve $(funn, F_g, F_b, \overrightarrow{\gamma_G}, \overrightarrow{\gamma_G}')$ as :

$$\overrightarrow{\gamma_G}'' = \begin{cases} \text{if } funn \in dom(F_g) & \text{, then } [\overrightarrow{\gamma_G}'[0], \overrightarrow{\gamma_G}[1]] \\ \text{otherwise} & \text{, then } \overrightarrow{\gamma_G}' \end{cases}$$

Where $\overrightarrow{\gamma_G}''$ is a list of two elements, index 0 is the global scope, and index 1 is the programmable block global scope, and we will retrieve it to be added as the resulted scope of the frame reduction. $\overrightarrow{\gamma_G}'$ is the scope that is resulted from the statement reduction in comp2. $\overrightarrow{\gamma_G}$ is the original starting state's global list of the frame reduction.

2.4 Architecture-Level Semantics

The architecture-level semantics is the topmost-level semantics, describing the entirety of the P4 pipeline from input packets to output packets, and it uses the statement semantics for reduction steps inside programmable blocks. The judgment form, shown at the top of Figure 7, consists of multiple components: starting from the left, an *architectural context* ctx_A (which contains the static componenents 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 process-

ing. 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) and the fixed-function block stages. 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. This is the block type (parser or control), the list of directed parameters, the block-local function map containing function declared inside the block, a list of declarations of variables done at a block-global level, a statement representing initialisations and instantiations of these block-global variables followed by the body (for parsers, a transition statement pointing to "start", for control blocks the content of the apply statement encased in a block), and the parser state map P between parser state names and their bodies (statements)[4] and a table map Tb between names of tables and tuples of expressions and matching kinds. Note that the parser state map is empty for control blocks, as is the table map for parser blocks.

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 parameters, a list of expression arguments, the architectural scope and the block type 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 parameters, the architectural scope, the block type and status to an updated architectural scope. Note that these functions are responsible for the copy-in copy-out behaviour of the *parseError* variable in parser blocks.

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 semantics), 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 globally-declared function names to tuples of function bodies (statements) and their directed parameter lists.

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

1. The *architectural environment* env_A, which in turn has four components:

[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 by convention

   (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 $\bar{\gamma}_G$, which contains 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 (as described in the statement semantics), or it is 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)

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 informs us of whether a parser state machine is finished, or the apply of a control block is finished.

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

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_PBL_INIT: when the pending architecture block is a programmable block, a new block-global scope γ' is created and initialised with the arguments of the programmable block (in the case of a parser block, *parseError* is also initially set to *NoError*). After this, the variables in the list of declarations are declared in γ'. The final result γ''' is appended to the top-level global scope $\bar{\gamma}_G[0]$, forming a new global scopes list $\bar{\gamma}'_G$. Also, the empty architecture frame list is changed to a single frame where the singleton statement list contains the initialisations of the parser block *stmt* followed by the body of the programmable block[5], the scopes stack contains a single empty scope and the current function name is the name of the programmable block.

3. 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.

[5] Conventions for how these are represented are mentioned in the explanation of B_p

4. 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}.

5. ARCH_PARSER_TRANS: In case the block at the block index i is a parser block and the current status is **tra** x', this rule will obtain the parser state body $stmt'$ of $P(x')$ (where P is the parser state map of the current parser block), and set the next frame to the singleton list of $stmt'$ with $[\gamma_\emptyset]$ as the scope stack and x' as the current function name, as well as set the status to **run**.

6. ARCH_PBL_EXEC describes a reduction step inside a programmable block: the first two premises are there to obtain the content of the statement semantics context. Then, the global block list, the frame list, the control plane configuration and the status are updated by a statement reduction step $\longrightarrow$.

7. ARCH_PBL_RET rule describes the final step of programmable block reduction that reduces to an empty frame list. The first two premises obtain the directed parameters of the block. If *state_fin* (termination conditions of the two programmable blocks) holds of the status and frame list (if status was **run**, it is then set to **tra** "*reject*" if we are in a parser block). The block output function out_p proceeds to update the architectural scope (notably dependent on architecture, it copies out the value of *parseError*). Furthermore, the architecture block list index is incremented by 1 and the block-global scope is dropped from the global scopes list.

$\boxed{ctx_A \vdash s_A \longrightarrow_A s_A'}$ architecture-level semantics

$$\frac{\begin{array}{l}\mathbf{inp} = \overline{ab}[i] \\ (\overline{io}'', \gamma_A') = in_A(\overline{io}, \gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i+1, \overline{io}'', \overline{io}', \gamma_A'), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run})} \text{ ARCH\_IN}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i] \\ pbl\_type((x_1, d_1), .., (x_n, d_n))F_b\, \overrightarrow{decl}\ stmt\ P\ Tb = B_p(f) \\ \gamma' = in_p((x_1, .., x_n), [d_1, .., d_n], [e_1, .., e_n], \gamma_A, pbl\_type) \\ \gamma'' = \text{replicate}(\overrightarrow{decl}, \gamma') \\ \gamma_G'' = \overrightarrow{\gamma_G}[0] \\ \overrightarrow{\gamma_G}' = [\gamma_G''] + +[\gamma'']\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}', [([stmt])^f_{[\gamma_\emptyset]}], C, \mathbf{run})} \text{ ARCH\_PBL\_INIT}$$

$$\frac{\begin{array}{l}\mathbf{ffbl}\, x = \overline{ab}[i] \\ ff = B_{ff}(x) \\ \gamma_A' = ff(\gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((i+1, \overline{io}, \overline{io}', \gamma_A'), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run})} \text{ ARCH\_FFBL}$$

$$\frac{\begin{array}{l}\mathbf{out} = \overline{ab}[i] \\ (\overline{io}'', \gamma_A') = out_A(\overline{io}', \gamma_A)\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run}) \longrightarrow_A ((0, \overline{io}, \overline{io}'', \gamma_A'), \overrightarrow{\gamma_G}, [\,]_A, C, \mathbf{run})} \text{ ARCH\_OUT}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, x(e_1, .., e_n) = \overline{ab}[i] \\ \mathbf{parser}((x_1, d_1), .., (x_n, d_n))F_b\, \overrightarrow{decl}\ stmt\ P\ Tb = B_p(x) \\ \text{not\_final\_state}(x') \\ stmt' = P(x')\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, \overrightarrow{\Phi}, C, \mathbf{tra}\ x') \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}', [([stmt'])^{x'}_{[\gamma_\emptyset]}], C', \mathbf{run})} \text{ ARCH\_PARSER\_TRANS}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, x(e_1, .., e_n) = \overline{ab}[i] \\ pbl\_type((x_1, d_1), .., (x_n, d_n))F_b\, \overrightarrow{decl}\ stmt\ P\ Tb = B_p(x) \\ (X, F_g, F_b, P, Tb) \vdash (\overrightarrow{\gamma_G}, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_\Phi (\overrightarrow{\gamma_G}', \overrightarrow{\Phi}', C', t')\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, \overrightarrow{\Phi}, C, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}', \overrightarrow{\Phi}', C', t')} \text{ ARCH\_PBL\_EXEC}$$

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i] \\ pbl\_type((x_1, d_1), .., (x_n, d_n))F_b\, \overrightarrow{decl}\ stmt\ P\ Tb = B_p(f) \\ \text{state\_fin}(t, \overrightarrow{\Phi}) \\ t' = \text{set\_fin\_status}(pbl\_type, t) \\ \gamma_A' = out_p(\overrightarrow{\gamma_G}, \gamma_A, [d_1, .., d_n], (x_1, .., x_n), pbl\_type, t')\end{array}}{(\overline{ab}, B_p, B_{ff}, in_A, out_A, in_p, out_p, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overrightarrow{\gamma_G}, \overrightarrow{\Phi}, C, t) \longrightarrow_A ((i+1, \overline{io}, \overline{io}', \gamma_A'), \text{take}(1, \overrightarrow{\gamma_G}), [\,]_A, C, \mathbf{run})} \text{ ARCH\_PBL\_RET}$$

Figure 7: Architecture-Level Semantics

A Concrete Syntax of Operations

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

Figure 8: P4 Unary Operations

The unary expressions included are shown in Figure 8. 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
	|	$\underline{\vee}$	bitwise xor
	|	|	bitwise or
	|	$\wedge$	binary and
	|	$\vee$	binary or

Figure 9: P4 Binary Operations

The binary expressions included are shown in Figure 8. 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 10. The statement semantics are shown in Figure 11.

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

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

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

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

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

Figure 10: Expression Reduction-of-Argument Semantics

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

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

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

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

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

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

Figure 11: Statement Reduction-of-Argument Semantics