The HOL4P4 P4 Formalization

Anoud Alshnakat
Didrik Lundberg

September 4, 2023

This is a description of the HOL4P4 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].

HOL4P4 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 and Values

x, f, tbl	string
b	boolean
bl	bit-string
i, w	natural number
m, n, o	indices

Figure 1: Variables

The (meta)variables shown in Figure 1 are standard designations for the data of P4 base types included in HOL4P4, plus the numerals i (sometimes w when referring to width) and the indices m, n, o which are not part of the P4 syntax, but used on a meta-level throughout this formalization. Depending on the context, strings are denoted with x (variable or parser state name), f (function or field name) or tbl (match-action table name). bl is a list of Boolean values, used to represent bit-strings of fixed width.

Figure 2 shows the values of our formalisation. This largely mirrors P4 base types , except for the lack of match kinds, arbitrary-sized constants and bit-strings of dynamically computed width. Notably, some derived types have also been placed here: headers, structs and extern object references as well as the $\perp$ value which is an artifact of our model. Other derived types (tuples, enums, header unions and stacks) are yet to be added to the model, while some (parser, control, package) are not treated as such here, since they cannot be manipulated at run time.

P4 types are sometimes explicitly referenced in the syntax, e.g. in declaration statements. The notation for this is shown in Figure 3. $struct\_ty$ can be either **header** or **struct**, and num_exp is an arithmetic expression of type integer. Regular string literals are the special case of an x_list with length 1.

v	::=	value
|	$boolv$	boolean value
|	$bitv$	bit-string
|	x	string literal
|	**struct** $\{x_1 = v_1; \ldots; x_n = v_n\}$	struct
|	**header** $boolv\{x_1 = v_1; \ldots; x_n = v_n\}$	header
|	**errmsg** x	error message
|	**ext_ref** i	extern object reference
|	**bot**	no value

Figure 2: P4 Values

τ	::=	type
|	**bool**	boolean
|	$\mathbf{bs}^{num\_exp}$	bit-string
|	$\bot$	no value
|	$struct\_ty[x_1\,\tau_1, .., x_n\,\tau_n]$	struct
|	x_list	string literal
|	**err**	error
|	**ext**	extern

Figure 3: Types

1.2 Expressions

HOL4P4 includes a subset of the full set of P4 expressions found in Section 8 of the P4 specification, shown in Figure 4.

e	::=	expression
|	v	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
|	**eHeader** $boolv\{x_1 = e_1; \ldots; x_n = e_n\}$	header
|	(e)	S

Figure 4: P4 Expressions

First, an expression can be a value v, the types of which were shown earlier in Figure 2. An expression can also be a variable, in which case it has an abstract variable name $varn$ which is either

the special "star" function return placeholder variable, in which case it holds the corresponding *funn*, or a regular variable, which holds the variable name x. Lists of expressions can be used in initialisation 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 values[1]. Bitstrings can be concatenated and sliced. The function call is built from the abstract function name *funn*, 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.

Furthermore, structs and headers can also have expression values assigned to their keys, and so be expressions (separate from the struct and header values).

1.3 Statements

stmt	::=		statement
	|	$\emptyset_{\text{stmt}}$	empty statement
	|	$lval := e$	assignment
	|	**if** e **then** $stmt_1$ **else** $stmt_2$	conditional
	|	$\{\overline{decl}\ stmt\}$	block
	|	**return** e	return
	|	$stmt_1; stmt_2$	sequence
	|	**verify** $e\ e'$	verify
	|	**transition** e	transition
	|	**apply** $tbl(e_1, .., e_n)$	apply
	|	■	extern

Figure 5: P4 Statements

`HOL4P4` includes a subset of the full set of P4 statements found in Section 11 of the P4 specification, shown in Figure 5. All of the statements in Figure 5 can be found in the specification, with the following exceptions: the block, verify, apply and extern statements. The block statement features an additional list of declarations (note the absence of declarations from this syntax, in contrast to the P4 specification), the apply statement (replacing the apply method in Section 11 of the P4 specification) features the list of arguments that are to be matched to the table *tbl* (which is possible to resolve at compile time), and the extern captures the semantics of an extern function or method. 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'.

*lval*s are shown in Figure 6 and 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]The concrete syntax of the many unary and binary operations is found in Appendix A

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

Figure 6: P4 l-values

1.4 Execution State

t	::=		execution status
	|	**run**	
	|	**ret** v	
	|	**tra** x	transition to parser state
s	::=		execution state
	|	$(\gamma_A, \overline{\gamma_G}, \overline{\Phi}, t)$ M	

Figure 7: P4 Execution State

The P4 execution state is shown in Figure 7. Note that a "P4 execution state" is not defined in the P4 specification, so it is entirely an artifact of the `HOL4P4` implementation. In short, the execution state s is a tuple of the architectural scope γ_A, the global scope list $\overline{\gamma}_G$, a frame list $\overline{\Phi}$, and the state status t.

Scopes

More formally, a scope $\gamma : X \hookrightarrow V * (X \cup \{\bot\})$ is a partial function from variable names $x \in X$ to tuples of their values $v \in V$ and optional variable names (used for variables introduced as out-directed function parameters). The following operations are be performed on γ in the semantics:

- dom(γ): 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 instead of its old value. By writing $\forall i \leq n.\ (x_i \mapsto v_i)\ \gamma$ we extend this to lists of mappings from variable names to values.

$\overline{\gamma}$ is a list of scopes $\gamma_1, \ldots, \gamma_n$ where the index 1 is the most recently entered scope, and n is the oldest.

Global scopes

The global scope list $\overline{\gamma_G}$ contains two elements; index 0 represents the programmable block local scope, while index 1 represents the global scope of the architecture. Initially, when a programmable block is entered, for each function declared globally in the architecture or locally in a programmable block a corresponding function return placeholder variable is declared in the local programmable block scope.

Lists of statements

The list of statements $\overline{stmt}$ is simply a list of $stmt_1, \ldots, stmt_n$ where $stmt_1$ belongs to the most recent block we have entered, and $stmt_n$ is the oldest one.

Whenever a block $\{\overline{decl}\ stmt_b\}; stmt$ is encountered, the block's statement $stmt_b$ will be appended to the old statement $stmt$ leaving an empty statement in its original place. So the next transition will become $stmt_b :: (\emptyset_{stmt}; stmt$

Function names

The abstract function name *funn* identifies a function name, extern function (using extern object name and function name) or extern constructor (using only extern name). In P4, functions can be declared globally, or locally in the programmable block (actions). We model the actions exactly as the functions, and to distinguish between them we store their signatures and bodies in two different locations: the programmable block-local functions' signatures and bodies in the mapping F_b, and the globally declared functions' and actions' signatures and bodies in the mapping F_g. Both F_b and F_g are stored in the context *ctx* which will be discussed later.

Frames

The frame Φ is a tuple of three members: a function name *funn*, a statement list $\overline{stmt}$ and a scope list $\overline{\gamma}$: together represented as $\overline{stmt}^{funn}_{\overline{\gamma}}$. Whenever a function call occurs, the frame of the callee will be appended to the list of frames in the state (in which the frame of the caller is the first) as $\Phi_{\text{callee}} :: \overline{\Phi}$.

Architectural scope

The architectural scope is of polymorphic type. For specific architectures, this is supposed to hold table configurations and the states extern objects, as well as any other stateful components of the architecture. Exactly how this is represented is entirely up to the architecture in question.

Status

Represented in the state with t. The status **run** represents that the program is executing under regular circumstances. **ret** v is used when the **return** statement returns a value v ($\perp$ if it is a void function) at the end of a function call. The status **tra** x signifies transition to a new parser state. The new parser state can be a final state in the case of **tra** *accept* or **tra** *reject*, or otherwise a state defined in the P4 program.

1.5 Context

The context *ctx* - separated out from the execution state since it remains invariant over reductions - is a tuple of the following:

1. *table_apply*: the apply table function which takes a table name, parameters to match, match kinds, default action with arguments and the architectural scope and returns an action with parameters.

2. X: the extern object map, which maps extern object names to tuples of constructors and their respective function maps.
3. F_g: the globally-declared function map. It maps global function and action names to tuples of their bodies, argument names and directions.
4. F_b: the programmable block-local function map. It maps local function names to tuples of their bodies, argument names and directions.
5. P: the parser states map. It maps the name of parser states to their bodies.
6. Tb: the table map. It maps the table name to its list of match kinds and default action with parameters.

2 Semantics

2.1 Expressions

The reductions of the expression semantics can not directly alter or have any side effects on the state. The rules reduce expressions - standard small step structural semantics - and may also produce a new frame (which can then indirectly have side effects on the state in other layers of the semantics), 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 scope (i.e. most recent scope γ that x is declared in). The evaluation will occur using the function lookup_v. It will sweep the list of scopes $\overline{\gamma}{+\!\!+}\overline{\gamma}_G$ and find the most recent (leftmost in the list) scope that contains variable x and this is because the scopes grow from right to left, in accordance with the description in Sections 6.8 and 10.2 of the P4 specification. The constant value of this variable is then propagated to the result of the expression reduction.

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

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. We fetch the parameter names and directions of the function using lookup_funn_sig, then each function's arguments will be checked against the direction in the same position. If the direction is in or none, then the argument will be reduced until it becomes a constant, otherwise if the argument has the direction out or inout it should not be reduced beyond variable lookup.

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**, *funn*) and a new frame. The new frame will be added by the enclosing statement semantic layer on top of the state's existing frames, as explained later. 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. *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 tuples of variable names and their directions $[(x_1, d_1), \ldots, (x_n, d_n)]$. Each argument will be checked to ensure that the arguments were reduced properly: (if d_i is in or none then the expression in the same position should be a constant, otherwise a variable). The new frame's scope γ' is a new fresh empty scope to which the parameters are declared and copied in to using the function copyin, depending on direction.

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

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

Headers and structs with expression fields

Structs or headers with expressions in place of values have their own reduction rules. The E_eSTRUCT rule will reduce the expression fields one at a time from left to right. Once all the expressions have become values we can transform the expression struct to a value struct via E_eSTRUCT_TO_V. Similar operations are applied on the header expressions.

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.

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.

Bit slicing

This reduction gives a bitvector *bitv*‴ that is reduced from the slicing operation. It extracts a contiguous list from the original *bitv* from the LSB *bitv*" to the MSB *bitv*'.

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

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

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

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

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

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

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

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

Concatenation

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

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

Unary and binary arithmetic operations

The unary and binary operations consist mostly of standard operations on bit vectors, and are for this reason elided in this section.

2.2 Statement Semantics

The semantics of the statements are presented in this section[2]. Note that the statement semantics operates on single frames.

Assignment

The assignment can assign to *lval*s (shown in Figure 6), which can be either variables 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.

$$\frac{\begin{array}{c}\overline{\gamma}' = (\overline{\gamma} + +\overline{\gamma_G})[lval \longmapsto v] \\ (\overline{\gamma_G}', \overline{\gamma}'') = \text{separate}(\overline{\gamma}')\end{array}}{ctx \vdash (\gamma_A, \overline{\gamma_G}, [([lval := v])^{funn}_{\overline{\gamma}}], \mathbf{run}) \rightarrow (\gamma_A, \overline{\gamma_G}', [([\emptyset_{\text{stmt}}])^{funn}_{\overline{\gamma}''}], \mathbf{run})} \quad \text{STMT\_ASS\_V}$$

If-then-else

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

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

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

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

Block

Once a block statement is encountered $\{\overline{decl}\ stmt\}$ the STMT_BLOCK_ENTER reduction will be used, which entails the $\overline{decl}$ being declared in a fresh scope that will be pushed to the local $\overline{\gamma}$ of the frame, and then the body $stmt$ of the block will be pushed to the statement list, in which the block statement has been reduced to the empty statement.

The STMT_BLOCK_EXEC rule simply describes small-step reduction of the block contents. We also check if the previous list $\overline{stmt}$ is empty or not: if it is empty, then the other single block statement rules can be applied directly. It is also important to check that the head of the list $stmt :: \overline{stmt}$ is not $\emptyset_{\text{stmt}}$, otherwise this rule is non-deterministic with STMT_BLOCK_EXIT.

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 $\emptyset_{\text{stmt}}$: it pops the head (containing only $\emptyset_{\text{stmt}}$) from the statement list as well as the scope corresponding to the block from the scope list $\overline{\gamma}$.

$$\frac{\begin{array}{l}\gamma = \text{replicate}(\overline{decl}) \\ \overline{\gamma}' = [\gamma] + +\overline{\gamma}\end{array}}{ctx \vdash (\gamma_A, \overline{\gamma_G}, [([\{\overline{decl}\ stmt\}])^{funn}_{\overline{\gamma}}], \textbf{run}) \rightarrow (\gamma_A, \overline{\gamma_G}, [(stmt :: [\emptyset_{\text{stmt}}])^{funn}_{\overline{\gamma}'}], \textbf{run})}\ \text{STMT\_BLOCK\_ENTER}$$

$$\frac{\begin{array}{l}\text{not\_empty } stmt \\ \text{not\_empty } \overline{stmt} \\ (apply\_table, X, F_g, F_b, P, Tb) \vdash (\gamma_A, \overline{\gamma_G}, [([stmt])^{funn}_{\overline{\gamma}}], t) \rightarrow (\gamma_A', \overline{\gamma_G}', \overline{\Phi}' +\!\!+ [(\overline{stmt}')^{funn}_{\overline{\gamma}'}], t')\end{array}}{(apply\_table, X, F_g, F_b, P, Tb) \vdash (\gamma_A, \overline{\gamma_G}, [(stmt :: \overline{stmt})^{funn}_{\overline{\gamma}}], t) \rightarrow (\gamma_A', \overline{\gamma_G}', \overline{\Phi}' +\!\!+ [(\overline{stmt}' + +\overline{stmt})^{funn}_{\overline{\gamma}'}], t')}\ \text{STMT\_BLO}$$

$$\frac{\begin{array}{l}\text{not\_empty } \overline{stmt} \\ \overline{\gamma}' = \text{tl } \overline{\gamma}\end{array}}{ctx \vdash (\gamma_A, \overline{\gamma_G}, [(\emptyset_{\text{stmt}} :: \overline{stmt})^{funn}_{\overline{\gamma}}], t) \rightarrow (\gamma_A, \overline{\gamma_G}, [(\overline{stmt})^{funn}_{\overline{\gamma}'}], t)}\ \text{STMT\_BLOCK\_EXIT}$$

Apply

The STMT_APPLY_V describes the apply table statement. Note that this is particular to our formalization: it is a method call in the P4 specification. Each apply statement has a table name tbl with a list of key expressions $e_1, \ldots, e_n$ to match on. This list has been previously reduced to constants one at a time in the rule STMT_APPLY_E. The Tb will return a list of match kinds $mk_1, \ldots, mk_n$ and a default action with arguments, which is then used with $apply\_table$ to perform the actual table matching.

Return

Once the expression of **return** is reduced to a constant value, the status is changed to **ret** v, and the statement becomes $\emptyset_{\text{stmt}}$. The rest of the function return procedure will be described in the sequence rule and frame-level semantics.

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

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

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

Sequence

The sequential statements rules STMT_SEQ1 and STMT_SEQ2 are standard. The status t must be **run** in the initial state. The non-standard element of STMT_SEQ1 is that if any new frame is created (e.g. by a function call) or a statement added to the statement list (e.g. by a block statement), 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 statements that follow sequentially can be discarded: this gives flexibility to have return or transition statements in the body rather than only at the end. Furthermore, STMT_SEQ1 and STMT_SEQ3 are both able to change the architectural scope.

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

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

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

Extern

The symbol ■ represents a statement capturing the inner semantics of an extern function: i.e. the semantics other than copy-in, copy-out and return behaviour. ■ is able to modify the architectural scope, the local scope list $\overline{\gamma}$ and the execution status (which is always set to **ret** v). The exact behaviour is determined by looking up the entry of the *funn* associated with the current frame in the extern function map X: since ■ is meant to be used only in extern function call bodies, *funn* is the name of the extern function that is currently being called.

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 ■.

$$\frac{\begin{array}{c} ext\_fun = \text{lookup\_ext\_fun}(funn, X) \\ (\gamma_A{}', \overline{\gamma}', v) = ext\_fun\,(\gamma_A, \overline{\gamma_G}, \overline{\gamma}) \end{array}}{(apply\_table, X, F_g, F_b, P, Tb) \vdash (\gamma_A, \overline{\gamma_G}, [([\blacksquare])^{funn}_{\overline{\gamma}}], \mathbf{run}) \rightarrow (\gamma_A{}', \overline{\gamma_G}, [([\emptyset_{\text{stmt}}])^{funn}_{\overline{\gamma}'}], \mathbf{ret}\ v)}\ \text{STMT\_EXT}$$

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.

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

Verify

The **verify** $e\,e'$ statement is used to check the whether the boolean expression e holds; if it does, then nothing happens, as stated 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".

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

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

2.3 Frame-Level Semantics

The statement semantics in the previous section operate on a single frame containing a list of statements, however when a function being called, a new frame will be pushed to the frame list of the state. The frame-level semantics will always try to execute the top frame, however FRAMES_COMP1 will be used when the status is not set to **ret** v and FRAMES_COMP2 when it is.

For FRAMES_COMP2, note that the the global scope list $\overline{\gamma}_G$ will contain a **var**(**star**, *funn*) declared for every callable function in the current programmable block (set in the architecture-level semantics). This variable is then updated with the value v that is found in the final status **ret** v of the statement semantics reduction. The copy-out function takes the callee function signature, the global scope list, the caller's scope list $\overline{\gamma}'$ and the callee's scope list $\overline{\gamma}''$ (note that $\overline{\gamma} = \overline{\gamma}''$ due to the semantics of **return**). It returns a new updated scopes list for both the global and caller's frame.

In the copyout function, each out-directed parameter should be copied out to the variable name of its argument: note that these argument names are stored in the callee's scope.

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

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

We define scopes_to_pass($funn, F_g, F_b, \overline{\gamma_G}$) for function names as:

$$\overline{\gamma_G}' = \begin{cases} \text{if } funn \in dom(F_b) & \text{, then } \overline{\gamma_G} \\ \text{otherwise} & \text{, then if } funn \in dom(F_g) \text{ then } [\emptyset_\gamma; \overline{\gamma_G}[1]] \text{ else } \overline{\gamma_G}\ , \end{cases}$$

and in the case *funn* is an extern function or constructor, $\overline{\gamma_G}' = [\emptyset_\gamma; \overline{\gamma_G}[1]]$. Recall that the global scope is at index 1 of $\overline{\gamma_G}$, and the programmable block-global scope is at index 0. $\overline{\gamma_G}'$ is propagated to the statement semantics as the global scope list of the initial state.

We define scopes_to_retrieve($funn, F_g, F_b, \overline{\gamma_G}, \overline{\gamma_G}'$) for function names as:

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

and in the case *funn* is an extern function or constructor, $\overline{\gamma_G}'' = [\overline{\gamma_G}[0]; \overline{\gamma_G}'[1]]$. The result here then becomes the global scope list of the final state of the frame-level reduction, after eventual copy-out.

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 frame semantics for reduction steps inside programmable blocks. The judgment form, shown at the top of Figure 8, 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 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) 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.

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.

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

[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 `HOL4P4` represents all parser state bodies as encased in a block by convention

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:
   (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 $\overline{\gamma_G}$, 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 ($[\,]_A$).
4. The status, which 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 premise 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 $\overline{\gamma_G}[0]$, forming a new global scopes list $\overline{\gamma_G}''$, in which initialise_var_stars initialises function return placeholder variables for all callable functions in the block. 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.

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

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.

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**, set_fin_status then sets it 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

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

$$\frac{\begin{array}{l}\mathbf{pbl}\, f(e_1, .., e_n) = \overline{ab}[i] \\ pbl\_type((x_1, d_1), .., (x_n, d_n)) F_b\, \overline{decl}\; stmt\; P\; Tb = B_p(f) \\ \gamma' = in_p((x_1, .., x_n), [d_1, .., d_n], [e_1, .., e_n], \gamma_A, pbl\_type) \\ \gamma'' = \mathrm{replicate}(\overline{decl}, \gamma') \\ \overline{\gamma_G}' = \mathrm{lastn}(1, \overline{\gamma_G}) \\ \overline{\gamma_G}'' = [\gamma''] + + \overline{\gamma_G}' \\ \overline{\gamma_G}''' = \mathrm{initialise\_var\_stars}(F_g, F_b, X, \overline{\gamma_G}'')\end{array}}{(\overline{ab}, B_p, B_{\mathrm{ff}}, in_A, out_A, in_p, out_p, apply\_table, X, F_g) \vdash ((i, \overline{io}, \overline{io}', \gamma_A), \overline{\gamma_G}, [\,]_A, \mathbf{run}) \longrightarrow_A ((i, \overline{io}, \overline{io}', \gamma_A), \overline{\gamma_G}''', [([stmt])^f_{[\gamma\emptyset]}], \mathbf{run})} \text{ ARCH\_PBL\_INIT}$$

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

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

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

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

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

Figure 8: Architecture-Level Semantics

A Concrete Syntax of Operations

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

Figure 9: P4 Unary Operations

The unary expressions included are shown in Figure 9. These include all of the unary operations in P4. Boolean negation is only defined on Booleans, the other operations have their standard meanings (note that unary plus is a no-op).

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

Figure 10: P4 Binary Operations

The binary expressions included are shown in Figure 9. These include all of the binary operations in P4.

B Semantics of Expression Reduction

This appendix describes semantics for reducing nested expressions. A selection of this type of expression semantics are shown in Figure 11, and for the statement semantics are in Figure 12.

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

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

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

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

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

Figure 11: Expression Reduction-of-Argument Semantics (selection)

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

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

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

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

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

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

Figure 12: Statement Reduction-of-Argument Semantics (selection)