P4 Specification

Release 0.1 (Auto-generated Draft 2024-06-21)

Anonymous Authors

Jun 21, 2024

Contents

CHAPTER 1

Introduction

This automatically generated document describes the syntax and semantics of P4_16. It defines the abstract syntax, runtime objects, and execution semantics in formal notation.

CHAPTER 2

Syntax

2.1 Conventions

$$
\begin{aligned}
\mathit{num} &::= \mathbb{N} \\
\mathit{boolean} &::= \mathbb{B} \\
\mathit{str} &::= \mathbb{T} \\
\mathit{path} &::= \mathit{str}^*
\end{aligned}
$$

Sets and maps are modeled as a list of elements, and a list of key-value pairs, respectively.

$$
\begin{aligned}
\mathit{set}(K) &::= K^* \\
\mathit{map}(K, V) &::= (K, V)^*
\end{aligned}
$$

2.2 Types

$$
\begin{aligned}
\mathit{type} \quad ::= \quad & \mathsf{void} \\
| \quad & \mathsf{bool} \\
| \quad & \mathsf{matchkind} \\
| \quad & \mathsf{err} \\
| \quad & \mathsf{str} \\
| \quad & \mathsf{int} \\
| \quad & \mathsf{int}\ \mathit{expr} \\
| \quad & \mathsf{bit}\ \mathit{expr} \\
| \quad & \mathsf{varbit}\ \mathit{expr} \\
| \quad & \mathsf{var}\ \mathit{name} \\
| \quad & \mathsf{spec}\ \mathit{name} < \mathit{type}^* > \\
| \quad & \mathsf{stack}\ \mathit{type}\ [\mathit{expr}] \\
| \quad & \mathsf{tuple}\ \mathit{type}^* \\
| \quad & \mathsf{any}
\end{aligned}
$$

2.3 Variables

Identifiers prefixed with a dot are always resolved in the top-level namespace.

$$
\begin{aligned}
id &::= str
\end{aligned}
$$

$$
\begin{aligned}
name &::= \ .\ id \\
&\quad\mid\ id
\end{aligned}
$$

2.4 Expressions

$$
\begin{aligned}
field &::= str
\end{aligned}
$$

$$
\begin{aligned}
expr ::=\ & boolean \\
\mid\ & str \\
\mid\ & \mathbb{N} \\
\mid\ & name \\
\mid\ & \{expr^*\} \\
\mid\ & \{(field, expr)^*\} \\
\mid\ & (unop\ expr) \\
\mid\ & (binop\ expr\ expr) \\
\mid\ & (expr^?\ expr\ :\ expr) \\
\mid\ & ((type)\ expr) \\
\mid\ & (expr\ \mathsf{mask}\ expr) \\
\mid\ & (expr \ldots expr) \\
\mid\ & expr\ [expr] \\
\mid\ & expr\ [expr\ :\ expr] \\
\mid\ & name.field \\
\mid\ & \mathsf{error}.field \\
\mid\ & expr.field \\
\mid\ & expr\ {<}type^*{>}\ (arg^*) \\
\mid\ & type\ (arg^*)
\end{aligned}
$$

The following operators are supported:

$$
\begin{aligned}
unop ::=\ & \mathsf{bnot} \mid \mathsf{lnot} \mid \mathsf{uminus} \\
binop ::=\ & \mathsf{plus} \mid \mathsf{splus} \mid \mathsf{minus} \mid \mathsf{sminus} \\
\mid\ & \mathsf{mul} \mid \mathsf{div} \mid \mathsf{mod} \mid \mathsf{shl} \mid \mathsf{shr} \\
\mid\ & \mathsf{le} \mid \mathsf{ge} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{eq} \mid \mathsf{ne} \\
\mid\ & \mathsf{band} \mid \mathsf{bxor} \mid \mathsf{bor} \\
\mid\ & \mathsf{concat} \mid \mathsf{land} \mid \mathsf{lor}
\end{aligned}
$$

2.5 Parameters and Arguments

Each parameter may be labeled with a direction.

$$
dir ::= \mathsf{no} \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{inout}
$$

$$
param ::= id\ dir\ type\ expr^?
$$

Functions and objects may have type parameters.

$$
tparam ::= id^*
$$

Arguments can be named.

$$
\begin{aligned}
arg ::=\ & expr \\
\mid\ & id = expr \\
\mid\ & \_
\end{aligned}
$$

2.6 Statements

$$\begin{array}{rcl} stmt & ::= & \mathsf{empty} \\ & | & expr := expr \\ & | & \mathsf{switch}\ (expr)\ swcase^* \\ & | & \mathsf{if}\ (expr)\ stmt\ stmt \\ & | & \{block\} \\ & | & \mathsf{exit} \\ & | & \mathsf{return}\ expr^? \\ & | & expr < type^* > (arg^*) \\ & | & \mathsf{transition}\ label \\ & | & \mathsf{select}\ (expr^*)\ selcase^* \\ & | & decl \end{array}$$

A block is a sequence of statements.

$$block \quad ::= \quad stmt^*$$

Switch statements may have fallthroughs or default cases.

$$\begin{array}{rcl} case & ::= & \mathsf{case}\ label \\ & | & \mathsf{case}\ label \\ & | & \mathsf{default} \\ swcase & ::= & case\ block \end{array}$$

Labels are used as names for parser states.

$$label \quad ::= \quad str$$

Select statements resemble switch statements but are used for describing parser state machine.

$$\begin{array}{rcl} mtch & ::= & \mathsf{case}\ expr \\ & | & \mathsf{default} \\ & | & \mathsf{case} \\ selcase & ::= & mtch^*\ str \end{array}$$

Statments carry signals, which represent non-local control flow such as return and exit.

$$sig \quad ::= \quad \mathsf{cont} \mid \mathsf{ret}\ val^? \mid \mathsf{exit}$$

2.7 Declarations

$$
\begin{array}{rcl}
\mathit{decl} & ::= & \mathsf{const}\ \mathit{type}\ \mathit{id} = \mathit{expr} \\
& \mid & \mathit{type}\ \mathit{id} = \mathit{expr}^{?} \\
& \mid & \mathit{type}\ \mathit{id}\ (\mathit{arg}^{*})\ \mathit{block}^{?} \\
& \mid & \mathsf{error}\ \mathit{field}^{*} \\
& \mid & \mathsf{matchkind}\ \mathit{field}^{*} \\
& \mid & \mathsf{struct}\ \mathit{id}\ (\mathit{field}, \mathit{type})^{*} \\
& \mid & \mathsf{header}\ \mathit{id}\ (\mathit{field}, \mathit{type})^{*} \\
& \mid & \mathsf{header\_union}\ \mathit{id}\ (\mathit{field}, \mathit{type})^{*} \\
& \mid & \mathsf{enum}\ \mathit{id}\ \mathit{field}^{*} \\
& \mid & \mathsf{enum}\ \mathit{id}\ \mathit{type}\ (\mathit{field}, \mathit{expr})^{*} \\
& \mid & \mathsf{type}\ \mathit{type}^{?}\ \mathit{decl}^{?}\ \mathit{id} \\
& \mid & \mathsf{typedef}\ \mathit{type}^{?}\ \mathit{decl}^{?}\ \mathit{id} \\
& \mid & \mathsf{value\_set}\ \{\mathit{type}\}\ (\mathit{expr})\ \mathit{id} \\
& \mid & \mathsf{parser}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*}) \\
& \mid & \mathsf{parser}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*})\ (\mathit{param}^{*})\ \mathit{decl}^{*}\ \mathit{state}^{*} \\
& \mid & \mathsf{action}\ \mathit{id}\ (\mathit{param}^{*})\ \mathit{block} \\
& \mid & \mathsf{table}\ \mathit{id}\ \mathit{key}^{*}\ \mathit{action}^{*}\ \mathit{entry}^{*}\ \mathit{default}^{?}\ \mathit{custom}^{*} \\
& \mid & \mathsf{control}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*}) \\
& \mid & \mathsf{control}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*})\ (\mathit{param}^{*})\ \mathit{decl}^{*}\ \mathit{block} \\
& \mid & \mathit{type}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*})\ \mathit{block} \\
& \mid & \mathsf{extern}\ \mathit{type}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*}) \\
& \mid & \mathit{id}\ (\mathit{param}^{*}) \\
& \mid & \mathsf{abstract}\ \mathit{type}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*}) \\
& \mid & \mathit{type}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*}) \\
& \mid & \mathsf{extern}\ <\mathit{id}>\ \mathit{tparam}^{*} \\
& \mid & \mathsf{package}\ \mathit{id}\ <\mathit{tparam}^{*}>\ (\mathit{param}^{*})
\end{array}
$$

A parser block specifies a finite state machine.

$$
\mathit{state} \quad ::= \quad \mathit{label}\ \mathit{block}
$$

A control block may contain match-action tables.

$$
\begin{array}{rcl}
\mathit{key} & ::= & \mathit{expr}\ \mathit{str} \\
\mathit{action} & ::= & \mathit{id}\ \mathit{arg}^{*} \\
\mathit{entry} & ::= & \mathit{mtch}^{*}\ \mathit{action} \\
\mathit{default} & ::= & \mathit{action}\ \mathit{boolean} \\
\mathit{custom} & ::= & \mathit{field}\ \mathit{expr}\ \mathit{boolean}
\end{array}
$$

2.8 Program

A program is a sequence of declarations.

$$
\mathit{program} \quad ::= \quad \mathit{decl}^{*}
$$

CHAPTER 3

Runtime

3.1 Runtime Types

Runtime representation of types, which are different from syntactic types.

$$
\begin{array}{rcl}
\mathit{rtype} & ::= & \mathsf{tbool} \\
& | & \mathsf{tint} \\
& | & \mathsf{tint}\ \mathit{num} \\
& | & \mathsf{tbit}\ \mathit{num} \\
& | & \mathsf{tvarbit}\ \mathit{num} \\
& | & \mathsf{tstr} \\
& | & \mathsf{terr}\ \mathit{field}^* \\
& | & \mathsf{tmatchkind}\ \mathit{field}^* \\
& | & \mathsf{tvar}\ \mathit{str} \\
& | & \mathsf{tnew}\ \mathit{str} \\
& | & \mathsf{tstack}\ \mathit{rtype}\ [\mathit{num}] \\
& | & \mathsf{ttuple}\ \mathit{rtype}^* \\
& | & \mathsf{tstruct}\ (\mathit{field}, \mathit{rtype})^* \\
& | & \mathsf{theader}\ (\mathit{field}, \mathit{rtype})^* \\
& | & \mathsf{tunion}\ (\mathit{field}, \mathit{rtype})^* \\
& | & \mathsf{tenum}\ \mathit{field}^* \\
& | & \mathsf{rt\_ref}
\end{array}
$$

Typedef environment is a map from type variable names to runtime types.

$$\mathit{tdenv} \quad ::= \quad \mathit{map}(\mathit{id}, \mathit{rtype})$$

Typedef visibility is a set of type variable names.

$$\mathit{tdvis} \quad ::= \quad \mathit{set}(\mathit{id})$$

3.2 Values

$$
\begin{array}{rcl}
val & ::= & \mathsf{vbool}\ boolean \\
& | & \mathsf{vint}\ num \\
& | & \mathsf{vint}\ num\ num \\
& | & \mathsf{varbit}\ num\ num \\
& | & \mathsf{vvbit}\ num\ num \\
& | & \mathsf{vstr}\ str \\
& | & \mathsf{verr}\ str \\
& | & \mathsf{vstack}\ val^*\ num\ num \\
& | & \mathsf{vtuple}\ val^* \\
& | & \mathsf{vstruct}\ (str, val)^* \\
& | & \mathsf{vheader}\ boolean\ (str, val)^* \\
& | & \mathsf{vunion}\ (str, val)^* \\
& | & \mathsf{venumfield}\ str \\
& | & \mathsf{vsenumfield}\ str\ val \\
& | & \mathsf{vref}\ str^*
\end{array}
$$

Environment is a map from variable names to values.

$$env \quad ::= \quad tdenv\ venv\ fenv$$

Visibility is a set of variable names.

$$vis \quad ::= \quad tdvis\ vvis\ fvis$$

3.3 Function

A P4 program may contain functions and methods for programmable/non-programmable objects. They are both called functions in this document, if the context is clear.

Most functions carry a visibility set, which is a set of type, variable, and function names that are visible to the function. Functions are like closures in the sense that they capture the environment in which they are defined. However, the visibility is captured instead of the environment itself, allowing mutation of the environment external to the function.

fmethod is a method of a programmable parser or control block. (which should be "apply") fexternmethod is a method of an external, fixed-function block. fmethod is a state of a finite state machine described by a programmable parser block. Note that fstate are mutually recursive, and it does not have a visibility set, for it inherits the visibility set of the parser block. ftable is a method of a match-action table. (which should be "apply")

$$
\begin{array}{rcl}
func & ::= & \mathsf{ffunc}\ vis\ tparam^*\ param^*\ type\ block \\
& | & \mathsf{fextern}\ vis\ tparam^*\ param^* \\
& | & \mathsf{fmethod}\ vis\ tparam^*\ param^*\ block \\
& | & \mathsf{fexternmethod}\ vis\ tparam^*\ param^* \\
& | & \mathsf{fstate}\ block \\
& | & \mathsf{faction}\ vis\ param^*\ block \\
& | & \mathsf{ftable}
\end{array}
$$

Function environment is a map from function names to functions.

$$fenv \quad ::= \quad map(id, func)$$

Function visibility is a set of function names.

$$fvis \quad ::= \quad set(id)$$

3.4 Objects

Objects are explicitly allocated at compile time via the instantiation phase. otable, ovalueset, and oextern are stateful, i.e., they retain information across invocations. oparser, ocontrol, and opackage are classified as objects, since they wrap around the stateful objects. Yet, a table need not be instantiated explicitly, where a declaration is considered as an instantiation.

$$
\begin{array}{rcl}
obj & ::= & \mathsf{ovalueset} \\
 & | & \mathsf{otable}\ key^*\ action^*\ entry^*\ default^?\ custom^*\ func \\
 & | & \mathsf{oextern}\ vis\ env \\
 & | & \mathsf{oparser}\ vis\ env\ func \\
 & | & \mathsf{ocontrol}\ vis\ env\ func \\
 & | & \mathsf{opackage}
\end{array}
$$

Store is a map from path to objects. Paths are fully-qualified names of objects, the local name of an object prepended with the fully qualified name of its enclosing namespace.

$$sto \quad ::= \quad map(path, obj)$$

3.5 Contexts

Context is a collection of environment and visibility information, to be used in the instantiation phase and the interpreter. Visibility restricts the environment to a specific set of mappings.

3.5.1 Context

A context in P4 is divided into three logical layers: global, object, and local.

$$
\begin{array}{rcl}
ctx & ::= & \{\mathsf{path}\ path,\ \mathsf{id}\ id, \\
 & & \ \mathsf{genv}\ genv,\ \mathsf{gvis}\ gvis, \\
 & & \ \mathsf{oenv}\ oenv,\ \mathsf{ovis}\ ovis, \\
 & & \ \mathsf{lenv}\ lenv\}
\end{array}
$$

Global and object layers have environment and visibility for typedefs, variables, and functions.

$$
\begin{array}{rcl}
genv & ::= & env \\
gvis & ::= & vis \\
oenv & ::= & env \\
ovis & ::= & vis
\end{array}
$$

Local layer has a typedef environment and a stack of environments, for each block scope. The topmost environment in the stack is the most recent environment. Local layer does not have a function environment, as function declarations are not allowed inside local blocks.

$$lenv \quad ::= \quad tdenv\ venv^*$$

3.5.2 Instantiation Context

Instantiation context is a subset of the context, used in the instantiation phase. It omits the local layer, as local blocks are not instantiated.

$$
\begin{array}{rcl}
ictx & ::= & \{\mathsf{genv}\ genv,\ \mathsf{gvis}\ gvis, \\
 & & \ \mathsf{oenv}\ oenv,\ \mathsf{ovis}\ ovis\}
\end{array}
$$

CHAPTER 4

Execution

4.1 Expressions

Expressions evaluate to a value, and may have side-effects on the evaluating context due to function calls.

4.1.1 Boolean

$$\frac{}{S\ C \vdash boolean : C\ (\mathsf{vbool}\ boolean)}$$

4.1.2 String

$$\frac{}{S\ C \vdash str : C\ (\mathsf{vstr}\ str)}$$

4.1.3 Number

4.1.4 Variable

If a variable is prefixed with a dot, it should be looked up in the top-level (global) context.

$$\frac{val = \mathrm{find}_{var_{glob}}(C, id)}{S\ C \vdash (.\ id) : C\ val}\ [\textsc{Interp\_expr-e\_var-top}] \qquad \frac{val = \mathrm{find}_{var}(C, id)}{S\ C \vdash (id) : C\ val}\ [\textsc{Interp\_expr-e\_var-bare}]$$

4.1.5 List

$$\frac{S\ C \vdash expr^* : C'\ val^*}{S\ C \vdash \{expr^*\} : C'\ (\mathsf{vtuple}\ val^*)}$$

4.1.6 Record

$$\frac{S\ C \vdash expr^* : C'\ val^*}{S\ C \vdash \{(str, expr)^*\} : C'\ (\mathsf{vstruct}\ (str, val)^*)}$$

4.1.7 Unary, Binary, and Ternary

$$\frac{\begin{array}{c} S\ C \vdash expr : C'\ val \\ val' = \mathrm{unop}(unop, val) \end{array}}{S\ C \vdash (unop\ expr) : C'\ val'}$$

$$\frac{\begin{array}{c} S\ C \vdash expr_l : C'\ val_l \\ S\ C' \vdash expr_r : C''\ val_r \\ val = \mathrm{binop}(binop, val_l, val_r) \end{array}}{S\ C \vdash (binop\ expr_l\ expr_r) : C''\ val}$$

$$\frac{\begin{array}{c} S\ C \vdash expr_c : C'\ val_c \\ \mathsf{vbool\ true} = \mathrm{cast}(val_c, \mathsf{tbool}) \\ S\ C \vdash expr_t : C''\ val \end{array}}{S\ C \vdash (expr_c^?\ expr_t\ :\ expr_f) : C''\ val} \qquad \frac{\begin{array}{c} S\ C \vdash expr_c : C'\ val_c \\ \mathsf{vbool\ false} = \mathrm{cast}(val_c, \mathsf{tbool}) \\ S\ C \vdash expr_f : C''\ val \end{array}}{S\ C \vdash (expr_c^?\ expr_t\ :\ expr_f) : C''\ val}$$

4.1.8 Cast

$$\frac{\begin{array}{c} C \vdash type : rtype \\ S\ C \vdash expr : C'\ val \\ val' = \mathrm{cast}(val, rtype) \end{array}}{S\ C \vdash ((type)\ expr) : C'\ val'}$$

4.1.9 Mask and Range

4.1.10 Accesses

Array Accesses

$$\frac{\begin{array}{c} S\ C \vdash expr_b : C'\ (\mathsf{vstack}\ val^*\ num_i\ num_s) \\ S\ C' \vdash expr_i : C''\ val_i \\ i = \mathrm{unpack}(val_i) \qquad val = val^*[i] \end{array}}{S\ C \vdash expr_b\ [expr_i] : C''\ val}$$

Bitstring Accesses

$$\frac{\begin{array}{c} S\ C \vdash expr_b : C'\ val_b \\ S\ C' \vdash expr_h : C''\ val_h \\ S\ C'' \vdash expr_l : C'''\ val_l \\ val = \text{bitslice}(val_b, val_h, val_l) \end{array}}{S\ C \vdash expr_b\ [expr_h\ :\ expr_l] : C'''\ val}$$

Type Accesses

$$\frac{\textsf{tenum}\ mems = \text{find}_{td_{glob}}(C, id)}{S\ C \vdash (.\ id).mem : C\ (\textsf{venumfield}\ mem)} \qquad \frac{\textsf{tenum}\ mems = \text{find}_{td}(C, id)}{S\ C \vdash (id).mem : C\ (\textsf{venumfield}\ mem)}$$

Error Accesses

$$\frac{\textsf{terr}\ mems = \text{find}_{td_{glob}}(C, \text{“}error''\text{)}}{S\ C \vdash \textsf{error}.mem : C\ (\textsf{verr}\ mem)}$$

Expression Accesses

$$\frac{S\ C \vdash expr_b : C'\ (\textsf{vheader}\ valid\ (str, val)^*) \qquad val = \text{find}_{field}((str, val)^*, mem)}{S\ C \vdash expr_b.mem : C'\ val} \qquad \frac{S\ C \vdash expr_b : C'\ (\textsf{vstruct}\ (str, val)^*)}{S\ C \vdash expr_b.m}$$

4.1.11 Call

Calls are handled by the rule Interp_call.

$$\frac{S\ C \vdash expr_f\ type^*\ arg^* : (\textsf{ret}\ val^?)\ C'}{S\ C \vdash expr_f\ <type^*>\ (arg^*) : C'\ val}$$

4.1.12 Interpreting a Sequence of Expressions

Rule Interp_expr is used to interpret a sequence of expressions.

$$\frac{}{S\ C \vdash \epsilon : C\ \epsilon}\ [\textsc{Interp\_exprs-base}] \qquad \frac{\begin{array}{c} S\ C \vdash expr : C'\ val \\ S\ C' \vdash expr^* : C''\ val^* \end{array}}{S\ C \vdash expr\ expr^* : C''\ (val\ val^*)}\ [\textsc{Interp\_exprs-rec}]$$

4.2 Statements

Statements are evaluated under a signal and a context. Signal represents the current non-local control flow. Statements evaluate to a signal and an updated context.

4.2.1 Empty

An empty statement does nothing.

$$\frac{}{S\ sig\ C \vdash \mathsf{empty} : sig\ C}$$

4.2.2 Assignment

An assignment evaluates the right-hand side expression and assigns it to the lvalue on the left-hand side.

$$\frac{\begin{array}{c} S\ C \vdash expr_r : C'\ val_r \\ S\ C' \vdash expr_l \in val_r \dashv C'' \end{array}}{S\ \mathsf{cont}\ C \vdash expr_l := expr_r : \mathsf{cont}\ C''}\ [\text{Interp\_stmt-i\_assign-cont}]$$

$$\frac{}{S\ sig\ C \vdash expr_l := expr_r : sig\ C}\ [\text{Interp\_stmt-i\_assign-nocont}]$$

4.2.3 Conditional

$$\frac{\begin{array}{c} S\ C \vdash expr_c : C'\ val_c \\ \mathsf{vbool\ true} = \mathrm{cast}(val_c, \mathsf{tbool}) \\ S\ \mathsf{cont}\ C' \vdash stmt_t : sig\ C'' \end{array}}{S\ \mathsf{cont}\ C \vdash \mathsf{if}\ (expr_c)\ stmt_t\ stmt_f : sig\ C''}\ [\text{Interp\_stmt-i\_if-cont-tru}]$$

$$\frac{\begin{array}{c} S\ C \vdash expr_c : C'\ val_c \\ \mathsf{vbool\ false} = \mathrm{cast}(val_c, \mathsf{tbool}) \\ S\ \mathsf{cont}\ C' \vdash stmt_f : sig\ C'' \end{array}}{S\ \mathsf{cont}\ C \vdash \mathsf{if}\ (expr_c)\ stmt_t\ stmt_f : sig\ C''}\ [\text{Interp\_stmt-i\_if-}$$

4.2.4 Block

$$\frac{S\ sig\ C \vdash block : sig'\ C'}{S\ sig\ C \vdash \{block\} : sig'\ C'}$$

4.2.5 Call

Calls are evaluated by the rule Interp_call. A return from the callee function is handled by the caller and produces a continue signal. Exit signals are propagated.

$$\frac{S\ C \vdash expr_f\ type^*\ arg^* : (\mathsf{ret}\ val)\ C'}{S\ \mathsf{cont}\ C \vdash expr_f < type^* > (arg^*) : \mathsf{cont}\ C'}\ [\text{Interp\_stmt-i\_call-cont-ret}]$$

$$\frac{S\ C \vdash expr_f\ type^*\ arg^* : \mathsf{cont}\ C'}{S\ \mathsf{cont}\ C \vdash expr_f < type^* > (arg^*) : \mathsf{cont}\ C'}\ [\text{Interp\_stm}$$

4.2.6 State Transition

State transitions in a parser state machine is handled by i_trans and i_select.

$$\frac{}{S\ \mathsf{cont}\ C \vdash \mathsf{transition}\ \text{“}accept'' : \mathsf{cont}\ C}\ [\text{Interp\_stmt-i\_trans-accept}]$$

$$\frac{}{S\ \mathsf{cont}\ C \vdash \mathsf{transition}\ \text{“}reject'' : \mathsf{cont}\ C}\ [\text{Interp\_stmt-i\_trans-re}$$

4.2.7 Declaration Statement

A declaration statement introduces a new variable in the context.

$$\frac{C \vdash \mathit{type} : \mathit{rtype} \quad S\ C \vdash \mathit{expr} : C'\ \mathit{val} \quad C'' = \mathrm{add}_{\mathit{var}_{loc}}(C', \mathit{id}, \mathit{rtype}, \mathit{val})}{S\ \mathsf{cont}\ C \vdash (\mathit{type}\ \mathit{id} = \mathit{expr}^{?}) : \mathsf{cont}\ C''}\ [\textsc{Interp\_stmt-i\_decl-cont-var-some}]$$

$$\frac{C \vdash \mathit{type} : \mathit{rtype} \quad \mathit{val} = \mathrm{default}(\mathit{rtype}) \quad C' = \mathrm{add}_{\mathit{var}_{loc}}(C, \mathit{id}, \mathit{rtype},}{S\ \mathsf{cont}\ C \vdash (\mathit{type}\ \mathit{id} = \epsilon) : \mathsf{cont}\ C'}$$

4.2.8 Switch

4.2.9 Exit

Exit statement immediately terminates the execution of all the blocks currently executing. Note that exit is not allowed within parser or functions.

$$\frac{}{S\ \mathsf{cont}\ C \vdash \mathsf{exit} : \mathsf{exit}\ C}\ [\textsc{Interp\_stmt-i\_exit-cont}] \qquad \frac{}{S\ \mathit{sig}\ C \vdash \mathsf{exit} : \mathit{sig}\ C}\ [\textsc{Interp\_stmt-i\_exit-nocont}]$$

4.2.10 Return

$$\frac{S\ C \vdash \mathit{expr} : C'\ \mathit{val}}{S\ \mathsf{cont}\ C \vdash \mathsf{return}\ \mathit{expr}^{?} : (\mathsf{ret}\ \mathit{val}^{?})\ C'}\ [\textsc{Interp\_stmt-i\_return-cont-some}] \qquad \frac{}{S\ \mathsf{cont}\ C \vdash \mathsf{return}\ \epsilon : (\mathsf{ret}\ \epsilon)\ C'}\ [\textsc{Interp\_stmt-i\_return-cont-non}$$

4.2.11 Interpreting a Block

A block is a sequence of statements.

$$\frac{}{S\ \mathit{sig}\ C \vdash \epsilon : \mathit{sig}\ C}\ [\textsc{Interp\_stmts-base-empty}] \qquad \frac{}{S\ \mathsf{ret}\ C \vdash \mathit{stmt}^{*} : \mathsf{ret}\ C}\ [\textsc{Interp\_stmts-base-ret}] \qquad \frac{}{S\ \mathsf{exit}\ C \vdash \mathit{stmt}^{*} : \mathsf{exit}\ C}\ [\textsc{Interp\_stm}$$

4.3 Calls

Function calls and method calls may both appear as expressions or statements in P4.

$$\frac{S\ C \vdash \mathit{name}\ \mathit{type}^{*}\ \mathit{arg}^{*} : \mathit{sig}\ C'}{S\ C \vdash (\mathit{name})\ \mathit{type}^{*}\ \mathit{arg}^{*} : \mathit{sig}\ C'}\ [\textsc{Interp\_call-func}] \qquad \frac{S\ C \vdash \mathit{expr}_b\ \mathit{mem}\ \mathit{type}^{*}\ \mathit{arg}^{*} : \mathit{sig}\ C'}{S\ C \vdash (\mathit{expr}_b.\mathit{mem})\ \mathit{type}^{*}\ \mathit{arg}^{*} : \mathit{sig}\ C'}\ [\textsc{Interp\_call-method}]$$

4.3.1 Finding the Callee Function and Determining the Callee Context

The first thing to do is to identify the callee function or method.

Function Calls

A function identifier may be prefixed with a dot, in which the function should be found in the toplevel (global) context. A function may be found in the object context, in case of an action call. Rule Interp_inter_call is used for calls that go beyond the caller's object boundary. In such case, the callee context first inherits the caller's global environment. Then, it will be restricted by the global visibility of the callee function afterwards. On the other hand, rule Interp_intra_call is used for calls that stay within the caller's object boundary.

$$\frac{\begin{array}{c} \mathit{func} = \mathrm{find}_{\mathit{func}_{\mathit{glob}}}(C_{\mathit{caller}}, \mathit{id}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{id} = \mathit{id}][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}] \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'} \end{array}}{S\ C_{\mathit{caller}} \vdash (.\ \mathit{id})\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'}}$$

$$\frac{\begin{array}{c} \mathit{func} = \mathrm{find}_{\mathit{func}_{\mathit{glob}}}(C_{\mathit{caller}}, \mathit{id}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{id} = \mathit{id}][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}] \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'} \end{array}}{S\ C_{\mathit{caller}} \vdash (\mathit{id})\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'}}$$

$$\frac{\begin{array}{c} \mathit{func} = \mathrm{find}_{\mathit{func}_{\mathit{obj}}}(C_{\mathit{caller}}, \mathit{id}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{id} = \mathit{id}][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}][.\mathsf{oenv} = C_{\mathit{caller}}.\mathsf{oenv}] \\ C_{\mathit{callee}'} = C_{\mathit{callee}}[.\mathsf{gvis} = C_{\mathit{caller}}.\mathsf{gvis}] \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}'} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'} \end{array}}{S\ C_{\mathit{caller}} \vdash (\mathit{id})\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}'}}$$

Method Calls

All method calls except "apply" on a table are inter-object calls. For inter-object calls, the callee context inherits the caller's global environment and takes the callee object's global visibility and object context.

$$\frac{\begin{array}{c} S\ C_{\mathit{caller}} \vdash \mathit{expr}_b : C_{\mathit{caller}'}\ (\mathsf{vref}\ \mathit{path}) \\ \mathsf{oextern}\ \mathit{vis}_{\mathit{glob}}\ \mathit{env}_{\mathit{obj}} = \mathrm{find}_{\mathit{obj}}(S, \mathit{path}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{path} = \mathit{path}][.\mathsf{id} = \mathit{mem}][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}][.\mathsf{oenv} = \mathit{env}_{\mathit{obj}}] \\ C_{\mathit{callee}'} = C_{\mathit{callee}}[.\mathsf{gvis} = \mathit{vis}_{\mathit{glob}}] \\ \mathit{func} = \mathrm{find}_{\mathit{func}_{\mathit{obj}}}(C_{\mathit{callee}'}, \mathit{mem}) \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}'} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''} \end{array}}{S\ C_{\mathit{caller}} \vdash \mathit{expr}_b\ \mathit{mem}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''}}$$

$$\frac{\begin{array}{c} S\ C_{\mathit{caller}} \vdash \mathit{expr}_b : C_{\mathit{caller}'}\ (\mathsf{vref}\ \mathit{path}) \\ \mathsf{oparser}\ \mathit{vis}_{\mathit{glob}}\ \mathit{env}_{\mathit{obj}}\ \mathit{func} = \mathrm{find}_{\mathit{obj}}(S, \mathit{path}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{path} = \mathit{path}][.\mathsf{id} = \text{“}\mathit{apply}''][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}][.\mathsf{oenv} = \mathit{env}_{\mathit{obj}}] \\ C_{\mathit{callee}'} = C_{\mathit{callee}}[.\mathsf{gvis} = \mathit{vis}_{\mathit{glob}}] \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}'} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''} \end{array}}{S\ C_{\mathit{caller}} \vdash \mathit{expr}_b\ \text{“}\mathit{apply}''\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''}}$$

$$\frac{\begin{array}{c} S\ C_{\mathit{caller}} \vdash \mathit{expr}_b : C_{\mathit{caller}'}\ (\mathsf{vref}\ \mathit{path}) \\ \mathsf{ocontrol}\ \mathit{vis}_{\mathit{glob}}\ \mathit{env}_{\mathit{obj}}\ \mathit{func} = \mathrm{find}_{\mathit{obj}}(S, \mathit{path}) \\ C_{\mathit{callee}} = \mathsf{new}[.\mathsf{path} = \mathit{path}][.\mathsf{id} = \text{“}\mathit{apply}''][.\mathsf{genv} = C_{\mathit{caller}}.\mathsf{genv}][.\mathsf{oenv} = \mathit{env}_{\mathit{obj}}] \\ C_{\mathit{callee}'} = C_{\mathit{callee}}[.\mathsf{gvis} = \mathit{vis}_{\mathit{glob}}] \\ S\ C_{\mathit{caller}}\ C_{\mathit{callee}'} \vdash \mathit{func}\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''} \end{array}}{S\ C_{\mathit{caller}} \vdash \mathit{expr}_b\ \text{“}\mathit{apply}''\ \mathit{type}^*\ \mathit{arg}^* : \mathit{sig}\ C_{\mathit{caller}''}}$$

$$\frac{\begin{array}{c} S\ C_{caller} \vdash expr_b : C_{caller'}\ (\mathsf{vref}\ path) \\ \mathsf{otable}\ key^*\ action^*\ entry^*\ default\ custom^*\ func = \mathrm{find}_{obj}(S, path) \\ C_{callee} = \mathsf{new}[.\mathsf{path} = path][.\mathsf{id} = \text{“}apply''][.\mathsf{genv} = C_{caller}.\mathsf{genv}][.\mathsf{oenv} = C_{caller}.\mathsf{oenv}] \\ C_{callee'} = C_{callee}[.\mathsf{gvis} = C_{caller}.\mathsf{gvis}] \\ S\ C_{caller}\ C_{callee'} \vdash func\ \epsilon\ \epsilon : sig\ C_{caller''} \end{array}}{S\ C_{caller} \vdash expr_b\ \text{“}apply''\ \epsilon\ \epsilon : sig\ C_{caller''}}$$

4.3.2 Passing Control to the Callee with More Restrictions on Visibility

Now, calls are classified as either inter-object or intra-object calls.

Calling Conventions

P4 adopts a copy-in/copy-out calling convention. Note that p4cherry defines two variants of copy-in. If a call is a “apply” method call on a parser or control object, the arguments are copied into the callee's object layer. (This is because the “apply” method of a parser and control scopes over the entire object.) Otherwise, the arguments are copied into the callee's local layer.

$copyin_{loc'}$ copies in a single parameter-value pair.

$$\begin{array}{lcll} \mathrm{copyin}_{loc'}(C, id\ dir\ type\ expr, val) & = & C' & \text{if } dir = \mathsf{out} \\ & & & \wedge\ C \vdash type : rtype \\ & & & \wedge\ val' = \mathrm{default}(rtype) \\ & & & \wedge\ C' = \mathrm{add}_{var_{loc}}(C, id, rtype, val') \\ \mathrm{copyin}_{loc'}(C, id\ dir\ type\ expr, val) & = & C' & \text{if } dir = \mathsf{no} \vee dir = \mathsf{in} \vee dir = \mathsf{inout} \\ & & & \wedge\ C \vdash type : rtype \\ & & & \wedge\ C' = \mathrm{add}_{var_{loc}}(C, id, rtype, val) \end{array}$$

$copyin_{loc}$ copies in a list of parameter-value pairs.

$$\begin{array}{lcll} \mathrm{copyin}_{loc}(C, \epsilon, \epsilon) & = & C & \\ \mathrm{copyin}_{loc}(C, param\ param^*, val\ val^*) & = & C'' & \text{if } C' = \mathrm{copyin}_{loc'}(C, param, val) \\ & & & \wedge\ C'' = \mathrm{copyin}_{loc}(C', param^*, val^*) \end{array}$$

$copyin_{obj'}$ and $copyin_{obj}$ are defined similarly.

$$\begin{array}{lcll} \mathrm{copyin}_{obj'}(C, id\ dir\ type\ expr, val) & = & C' & \text{if } dir = \mathsf{out} \\ & & & \wedge\ C \vdash type : rtype \\ & & & \wedge\ val' = \mathrm{default}(rtype) \\ & & & \wedge\ C' = \mathrm{add}_{var_{obj}}(C, id, rtype, val') \\ \mathrm{copyin}_{obj'}(C, id\ dir\ type\ expr, val) & = & C' & \text{if } dir = \mathsf{no} \vee dir = \mathsf{in} \vee dir = \mathsf{inout} \\ & & & \wedge\ C \vdash type : rtype \\ & & & \wedge\ C' = \mathrm{add}_{var_{obj}}(C, id, rtype, val) \end{array}$$

$$\begin{array}{lcll} \mathrm{copyin}_{obj}(C, \epsilon, \epsilon) & = & C & \\ \mathrm{copyin}_{obj}(C, param\ param^*, val\ val^*) & = & C'' & \text{if } C' = \mathrm{copyin}_{obj'}(C, param, val) \\ & & & \wedge\ C'' = \mathrm{copyin}_{obj}(C', param^*, val^*) \end{array}$$

Relation copyouts defines the copy-out operation.

$$\begin{array}{llcll} [\textsc{Copyout-do}] & S\ C_{caller}\ C_{callee}\ param\ expr & \hookrightarrow & C_{caller'} & \text{if } param = id\ dir\ type\ expr' \\ & & & & \wedge\ dir = \mathsf{inout} \vee dir = \mathsf{out} \\ & & & & \wedge\ val = \mathrm{find}_{var}(C_{callee}, id) \\ & & & & \wedge\ S\ C_{caller} \vdash expr \in val \dashv C_{caller'} \\ [\textsc{Copyout-pass}] & S\ C_{caller}\ C_{callee}\ param\ expr & \hookrightarrow & C_{caller} & \text{if } param = id\ dir\ type\ expr' \\ & & & & \wedge\ dir = \mathsf{no} \vee dir = \mathsf{in} \end{array}$$

$$\begin{array}{llcll} [\textsc{Copyouts-base}] & S\ C_{caller}\ C_{callee}\ \epsilon\ \epsilon & \hookrightarrow & C_{caller} & \\ [\textsc{Copyouts-rec}] & S\ C_{caller}\ C_{callee}\ (param\ param^*)\ (expr\ expr^*) & \hookrightarrow & C_{caller''} & \text{if } S\ C_{caller}\ C_{callee}\ param\ expr \hookrightarrow C_{caller'} \\ & & & & \wedge\ S\ C_{caller'}\ C_{callee}\ param^*\ expr^* \hookrightarrow C_{caller''} \end{array}$$

Intra-object Calls

For intra-object calls, after evaluating the callee's body and copy-out, the caller context inherits the callee's object environment to take the mutations of object-local variables into account.

$$\frac{\begin{array}{c} C_{callee'} = C_{callee}[\mathsf{.ovis} = vis] \qquad C_{callee''} = \mathsf{enter}\ C_{callee'} \\ (param'^*, expr^*) = \mathrm{align}_{args}(param^*, arg^*) \\ S\ C_{caller} \vdash expr^* : C_{caller'}\ val^* \\ C_{callee'''} = \mathrm{copyin}_{loc}(C_{callee''}, param^*, val^*) \\ S\ \mathsf{cont}\ C_{callee'''} \vdash block : sig\ C_{callee''''} \\ S\ C_{caller'}\ C_{callee''''}\ param^*\ expr^* \hookrightarrow C_{caller''} \\ C_{caller'''} = C_{caller''}[\mathsf{.oenv} = C_{callee''''}\mathsf{.oenv}] \end{array}}{S\ C_{caller}\ C_{callee} \vdash (\mathsf{faction}\ vis\ param^*\ block)\ type^*\ arg^* : sig\ C_{caller'''}}$$

Inter-object Calls

Since all globals are immutable in P4, the caller does not need to inherit the callee's global environment after callee evaluation.

$$\frac{\begin{array}{c} C_{callee'} = C_{callee}[\mathsf{.gvis} = vis_{glob}] \qquad C_{callee''} = \mathsf{enter}\ C_{callee'} \\ (param'^*, expr^*) = \mathrm{align}_{args}(param^*, arg^*) \\ S\ C_{caller} \vdash expr^* : C_{caller'}\ val^* \\ C_{callee'''} = \mathrm{copyin}_{loc}(C_{callee''}, param^*, val^*) \\ S\ C_{callee'''} \vdash id : sig\ C_{callee''''} \\ S\ C_{caller'}\ C_{callee'''}\ param^*\ expr^* \hookrightarrow C_{caller''} \end{array}}{S\ C_{caller}\ C_{callee} \vdash (\mathsf{fextern}\ vis_{glob}\ tparam^*\ param^*)\ type^*\ arg^* : sig\ C_{caller''}}$$

$$\frac{\begin{array}{c} C_{callee'} = C_{callee}[\mathsf{.ovis} = vis_{obj}] \qquad C_{callee''} = \mathsf{enter}\ C_{callee'} \\ (param'^*, expr^*) = \mathrm{align}_{args}(param^*, arg^*) \\ S\ C_{caller} \vdash expr^* : C_{caller'}\ val^* \\ C_{callee'''} = \mathrm{copyin}_{loc}(C_{callee''}, param^*, val^*) \\ S\ C_{callee'''} \vdash id : sig\ C_{callee''''} \\ S\ C_{caller'}\ C_{callee'''}\ param^*\ expr^* \hookrightarrow C_{caller''} \end{array}}{S\ C_{caller}\ C_{callee} \vdash (\mathsf{fexternmethod}\ vis_{obj}\ tparam^*\ param^*)\ type^*\ arg^* : sig\ C_{caller''}}$$

$$\frac{\begin{array}{c} C_{callee'} = C_{callee}[\mathsf{.ovis} = vis_{obj}] \\ (param'^*, expr^*) = \mathrm{align}_{args}(param^*, arg^*) \\ S\ C_{caller} \vdash expr^* : C_{caller'}\ val^* \\ C_{callee''} = \mathrm{copyin}_{obj}(C_{callee'}, param^*, val^*) \\ S\ \mathsf{cont}\ C_{callee''} \vdash block : sig\ C_{callee'''} \\ S\ C_{caller'}\ C_{callee'''}\ param^*\ expr^* \hookrightarrow C_{caller''} \end{array}}{S\ C_{caller}\ C_{callee} \vdash (\mathsf{fmethod}\ vis_{obj}\ tparam^*\ param^*\ block)\ type^*\ arg^* : sig\ C_{caller''}}$$

$$\frac{\begin{array}{c} C_{callee'} = C_{callee}[\mathsf{.gvis} = vis] \qquad C_{callee''} = \mathsf{enter}\ C_{callee'} \\ (param'^*, expr^*) = \mathrm{align}_{args}(param^*, arg^*) \\ S\ C_{caller} \vdash expr^* : C_{caller'}\ val^* \\ C_{callee'''} = \mathrm{copyin}_{loc}(C_{callee''}, param^*, val^*) \\ S\ \mathsf{cont}\ C_{callee'''} \vdash block : sig\ C_{callee''''} \\ S\ C_{caller'}\ C_{callee''''}\ param^*\ expr^* \hookrightarrow C_{caller''} \end{array}}{S\ C_{caller}\ C_{callee} \vdash (\mathsf{faction}\ vis\ param^*\ block)\ type^*\ arg^* : sig\ C_{caler''}}$$