

---

# **P4 Specification**

***Release 0.0 (Auto-generated Draft 2024-06-18)***

**Anonymous Authors**

**Jun 18, 2024**



---

## Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Syntax</b>	<b>3</b>
2.1	Conventions . . . . .	3
2.2	Types . . . . .	3
2.3	Variables . . . . .	4
2.4	Expressions . . . . .	4
2.5	Parameters and Arguments . . . . .	4
2.6	Statements . . . . .	5
2.7	Declarations . . . . .	6
2.8	Program . . . . .	6
<b>3</b>	<b>Runtime</b>	<b>7</b>
3.1	Runtime Types . . . . .	7
3.2	Values . . . . .	8
3.3	Function . . . . .	8
3.4	Objects . . . . .	9
3.5	Contexts . . . . .	9
<b>4</b>	<b>Execution</b>	<b>11</b>
4.1	Expressions . . . . .	11
4.2	Statements . . . . .	13
4.3	Calls . . . . .	15



# 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.



## 2.1 Conventions

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

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

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

## 2.2 Types

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

## 2.3 Variables

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

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

## 2.4 Expressions

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

The following operators are supported:

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

## 2.5 Parameters and Arguments

Each parameter may be labeled with a direction.

$$\begin{aligned} dir &::= \text{no} \mid \text{in} \mid \text{out} \mid \text{inout} \\ param &::= id\ dir\ type\ expr^? \end{aligned}$$

Functions and objects may have type parameters.

$$tparam \quad ::= \quad id^*$$

Arguments can be named.

$$\begin{aligned} arg &::= expr \\ &\quad | id = expr \\ &\quad | - \end{aligned}$$



## 2.6 Statements

```

stmt ::= empty
      | expr := expr
      | switch (expr) swcase*
      | if (expr) stmt stmt
      | {block}
      | exit
      | return expr?
      | expr {type*} (arg*)
      | transition label
      | select (expr*) selcase*
      | decl

```

A block is a sequence of statements.

```
block ::= stmt*
```

Switch statements may have fallthroughs or default cases.

```

case ::= case label
      | case label
      | default
swcase ::= case block

```

Labels are used as names for parser states.

```
label ::= str
```

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

```

mtch ::= case expr
      | default
      | case
selcase ::= mtch* str

```

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

```
sig ::= cont | ret val? | exit
```

## 2.7 Declarations

```

decl ::=
  | const type id = expr
  | type id = expr?
  | type id (arg*) block?
  | error field*
  | matchkind field*
  | struct id (field, type)*
  | header id (field, type)*
  | header_union id (field, type)*
  | enum id field*
  | enum id type (field, expr)*
  | type type? decl? id
  | typedef type? decl? id
  | value_set {type} (expr) id
  | parser id {tparam*} (param*)
  | parser id {tparam*} (param*) (param*) decl* state*
  | action id (param*) block
  | table id key* entry* default? custom*
  | control id {tparam*} (param*)
  | control id {tparam*} (param*) (param*) decl* block
  | type id {tparam*} (param*) block
  | extern type id {tparam*} (param*)
  | id (param*)
  | abstract type id {tparam*} (param*)
  | type id {tparam*} (param*)
  | extern {id} tparam*
  | package id {tparam*} (param*)

```

A parser block specifies a finite state machine.

```
state ::= label block
```

A control block may contain match-action tables.

```

key   ::= expr str
action ::= id arg*
entry  ::= mtch* action
default ::= action boolean
custom ::= field expr boolean

```

## 2.8 Program

A program is a sequence of declarations.

```
program ::= decl*
```

### 3.1 Runtime Types

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

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

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

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

Typedef visibility is a set of type variable names.

$$tdvis ::= \text{set}(id)$$

## 3.2 Values

```

val ::= vbool boolean
      | vint num
      | vint num num
      | varbit num num
      | vvbit num num
      | vstr str
      | verr str
      | vstack val* num num
      | vtuple val*
      | vstruct (str, val)*
      | vheader boolean (str, val)*
      | vunion (str, val)*
      | venumfield str
      | vsenumfield str val
      | vref str*

```

Environment is a map from variable names to values.

```
env ::= tdenv venv fenv
```

Visibility is a set of variable names.

```
vis ::= 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”)

```

func ::= ffunc vis tparam* param* type block
      | fextern str vis tparam* param*
      | fmethod vis tparam* param* block
      | fexternmethod str vis tparam* param*
      | fstate block
      | faction vis param* block
      | ftable

```

Function environment is a map from function names to functions.

```
fenv ::= map(id, func)
```

Function visibility is a set of function names.

```
fvis ::= 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.

```

obj ::= ovalueset
      | otable key* action* entry* default? custom* func
      | oextern vis env
      | oparser vis env func
      | ocontrol vis env func
      | opackage

```

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 ::= 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.

```

ctx ::= {genv genv, gvis gvis,
         oenv oenv, ovis ovis,
         lenv lenv}

```

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

```

genv ::= env
gvis ::= vis
oenv ::= env
ovis ::= vis

```

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 ::= 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.

```

ictx ::= {genv genv, gvis gvis,
          oenv oenv, ovis ovis}

```



### 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

$$\overline{S \ C \vdash \text{boolean} : C \ (\text{vbool } \text{boolean})}$$

#### 4.1.2 String

$$\overline{S \ C \vdash \text{str} : C \ (\text{vstr } \text{str})}$$

#### 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.

$$\frac{val = \text{find}_{var_{glob}}(C, id)}{S \ C \vdash (. \ id) : C \ val} [\text{INTERP\_EXPR-E\_VAR-TOP}] \quad \frac{val = \text{find}_{var}(C, id)}{S \ C \vdash (id) : C \ val} [\text{INTERP\_EXPR-E\_VAR-BARE}]$$

### 4.1.5 List

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

### 4.1.6 Record

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

### 4.1.7 Unary, Binary, and Ternary

$$\frac{S \ C \vdash \text{expr} : C' \ \text{val} \quad \text{val}' = \text{unop}(\text{unop}, \text{val})}{S \ C \vdash (\text{unop } \text{expr}) : C' \ \text{val}'}$$

$$\frac{S \ C \vdash \text{expr}_l : C' \ \text{val}_l \quad S \ C' \vdash \text{expr}_r : C'' \ \text{val}_r \quad \text{val} = \text{binop}(\text{binop}, \text{val}_l, \text{val}_r)}{S \ C \vdash (\text{binop } \text{expr}_l \ \text{expr}_r) : C'' \ \text{val}}$$

$$\frac{S \ C \vdash \text{expr}_c : C' \ \text{val}_c \quad \text{vbool true} = \text{cast}(\text{val}_c, \text{tbool}) \quad S \ C \vdash \text{expr}_t : C'' \ \text{val}}{S \ C \vdash (\text{expr}_c^? \ \text{expr}_t : \ \text{expr}_f) : C'' \ \text{val}} \quad \frac{S \ C \vdash \text{expr}_c : C' \ \text{val}_c \quad \text{vbool false} = \text{cast}(\text{val}_c, \text{tbool}) \quad S \ C \vdash \text{expr}_f : C'' \ \text{val}}{S \ C \vdash (\text{expr}_c^? \ \text{expr}_t : \ \text{expr}_f) : C'' \ \text{val}}$$

### 4.1.8 Cast

$$\frac{C \vdash \text{type} : \text{rtype} \quad S \ C \vdash \text{expr} : C' \ \text{val} \quad \text{val}' = \text{cast}(\text{val}, \text{rtype})}{S \ C \vdash ((\text{type}) \ \text{expr}) : C' \ \text{val}'}$$

### 4.1.9 Mask and Range

#### 4.1.10 Accesses

##### Array Accesses

$$\frac{S \ C \vdash \text{expr}_b : C' \ (\text{vstack } \text{val}^* \ \text{num}_i \ \text{num}_s) \quad S \ C' \vdash \text{expr}_i : C'' \ \text{val}_i \quad i = \text{unpack}(\text{val}_i) \quad \text{val} = \text{val}^*[i]}{S \ C \vdash \text{expr}_b \ [\text{expr}_i] : C'' \ \text{val}}$$



## Bitstring Accesses

$$\frac{\begin{array}{l} S \ C \vdash \text{expr}_b : C' \ \text{val}_b \\ S \ C' \vdash \text{expr}_h : C'' \ \text{val}_h \\ S \ C'' \vdash \text{expr}_l : C''' \ \text{val}_l \\ \text{val} = \text{bitslice}(\text{val}_b, \text{val}_h, \text{val}_l) \end{array}}{S \ C \vdash \text{expr}_b [\text{expr}_h : \text{expr}_l] : C''' \ \text{val}}$$

## Type Accesses

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

## Error Accesses

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

## Expression Accesses

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

### 4.1.11 Call

Calls are handled by the rule `Interp_call`.

$$\frac{S \ C \vdash \text{expr}_f \ \text{type}^* \ \text{arg}^* : (\text{ret val}^?) \ C'}{S \ C \vdash \text{expr}_f \ \{\text{type}^*\} \ (\text{arg}^*) : C' \ val}$$

### 4.1.12 Interpreting a Sequence of Expressions

Rule `Interp_expr` is used to interpret a sequence of expressions.

$$\frac{}{S \ C \vdash \epsilon : C \ \epsilon} [\text{INTERP_EXPRS-BASE}] \quad \frac{\begin{array}{l} S \ C \vdash \text{expr} : C' \ val \\ S \ C' \vdash \text{expr}^* : C'' \ val^* \end{array}}{S \ C \vdash \text{expr} \ \text{expr}^* : C'' \ (val \ val^*)} [\text{INTERP_EXPRS-REC}]$$

## 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.

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

### 4.2.2 Assignment

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

$$\frac{\begin{array}{l} S C \vdash \text{expr}_r : C' \text{ val}_r \\ S C' \vdash \text{expr}_l \in \text{val}_r \dashv C'' \end{array}}{S \text{ cont } C \vdash \text{expr}_l := \text{expr}_r : \text{cont } C''} [\text{INTERP\_STMT-I\_ASSIGN-CONT}] \quad \frac{}{S \text{ sig } C \vdash \text{expr}_l := \text{expr}_r : \text{sig } C} [\text{INTERP\_STMT-I\_ASSIGN-NOCONT}]$$

### 4.2.3 Conditional

$$\frac{\begin{array}{l} S C \vdash \text{expr}_c : C' \text{ val}_c \\ \text{vbool true} = \text{cast}(\text{val}_c, \text{tbool}) \\ S \text{ cont } C' \vdash \text{stmt}_t : \text{sig } C'' \end{array}}{S \text{ cont } C \vdash \text{if } (\text{expr}_c) \text{ stmt}_t \text{ stmt}_f : \text{sig } C''} [\text{INTERP\_STMT-I\_IF-CONT-TRU}] \quad \frac{\begin{array}{l} S C \vdash \text{expr}_c : C' \text{ val}_c \\ \text{vbool false} = \text{cast}(\text{val}_c, \text{tbool}) \\ S \text{ cont } C' \vdash \text{stmt}_f : \text{sig } C'' \end{array}}{S \text{ cont } C \vdash \text{if } (\text{expr}_c) \text{ stmt}_t \text{ stmt}_f : \text{sig } C''} [\text{INTERP\_STMT-I\_IF-CONT-FAL}]$$

### 4.2.4 Block

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

### 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.

$$\frac{S C \vdash \text{expr}_f \text{ type}^* \text{ arg}^* : (\text{ret val}) C'}{S \text{ cont } C \vdash \text{expr}_f \{ \text{type}^* \} (\text{arg}^*) : \text{cont } C'} [\text{INTERP\_STMT-I\_CALL-CONT-RET}] \quad \frac{S C \vdash \text{expr}_f \text{ type}^* \text{ arg}^* : \text{cont } C'}{S \text{ cont } C \vdash \text{expr}_f \{ \text{type}^* \} (\text{arg}^*) : \text{cont } C'} [\text{INTERP\_STMT-I\_CALL-CONT-RET}]$$

### 4.2.6 State Transition

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

$$\frac{}{S \text{ cont } C \vdash \text{transition "accept"} : \text{cont } C} [\text{INTERP\_STMT-I\_TRANS-ACCEPT}] \quad \frac{}{S \text{ cont } C \vdash \text{transition "reject"} : \text{cont } C} [\text{INTERP\_STMT-I\_TRANS-REJECT}]$$

## 4.2.7 Declaration Statement

A declaration statement introduces a new variable in the context.

$$\frac{C \vdash type : rtype \quad S \ C \vdash expr : C' \ val \quad C'' = \text{add}_{var_{loc}}(C', id, rtype, val)}{S \text{ cont } C \vdash (type \ id = expr^?) : \text{cont } C''} [\text{INTERP\_STMT-I\_DECL-CONT-VAR-SOME}] \quad \frac{C \vdash type : rtype \quad val = \text{default}(rtype) \quad C' = \text{add}_{var_{loc}}(C, id, rtype, val)}{S \text{ cont } C \vdash (type \ id = \epsilon) : \text{cont } C'}$$

## 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.

$$\frac{}{S \text{ cont } C \vdash \text{exit} : \text{exit } C} [\text{INTERP\_STMT-I\_EXIT-CONT}] \quad \frac{}{S \ sig \ C \vdash \text{exit} : sig \ C} [\text{INTERP\_STMT-I\_EXIT-NOCONT}]$$

## 4.2.10 Return

$$\frac{S \ C \vdash expr : C' \ val}{S \text{ cont } C \vdash \text{return } expr^? : (\text{ret } val^?) \ C'} [\text{INTERP\_STMT-I\_RETURN-CONT-SOME}] \quad \frac{}{S \text{ cont } C \vdash \text{return } \epsilon : (\text{ret } \epsilon) \ C'} [\text{INTERP\_STMT-I\_RETURN-CONT-NON}]$$

## 4.2.11 Interpreting a Block

A block is a sequence of statements.

$$\frac{}{S \ sig \ C \vdash \epsilon : sig \ C} [\text{INTERP\_STMTS-BASE-EMPTY}] \quad \frac{}{S \ ret \ C \vdash stmt^* : \text{ret } C} [\text{INTERP\_STMTS-BASE-RET}] \quad \frac{}{S \ exit \ C \vdash stmt^* : \text{exit } C} [\text{INTERP\_STMTS-BASE-EXIT}]$$

## 4.3 Calls

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

$$\frac{S \ C \vdash name \ type^* \ arg^* : sig \ C'}{S \ C \vdash (name) \ type^* \ arg^* : sig \ C'} [\text{INTERP\_CALL-FUNC}] \quad \frac{S \ C \vdash expr_b \ mem \ type^* \ arg^* : sig \ C'}{S \ C \vdash (expr_b.mem) \ type^* \ arg^* : sig \ C'} [\text{INTERP\_CALL-METHOD}]$$

### 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's context first inherits the caller's global context. Then, it will be restricted by the global visibility of the callee function. On the other hand, rule `Interp_intra_call` is used for calls that stay within the caller's object boundary.

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

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

$$\frac{\begin{array}{l} func = \text{find}_{func_{obj}}(C_{caller}, id) \\ S C_{caller} \vdash func \text{ type}^* \text{ arg}^* : sig C_{callee'} \end{array}}{S C_{caller} \vdash (id) \text{ type}^* \text{ arg}^* : sig C_{callee'}}$$

## Method Calls

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

$$\frac{\begin{array}{l} S C_{caller} \vdash expr_b : C_{callee'} \text{ (vref path)} \\ \text{oextern } vis_{glob} \text{ env}_{obj} = \text{find}_{obj}(S, path) \\ C_{callee} = \text{new}[\text{.genv} = C_{caller}.\text{genv}][\text{.oenv} = \text{env}_{obj}] \quad C_{callee'} = \text{res\_glob } C_{callee} \text{ by } vis_{glob} \\ func = \text{find}_{func_{obj}}(C_{callee'}, mem) \\ S C_{caller} C_{callee'} \vdash func \text{ type}^* \text{ arg}^* : sig C_{callee''} \end{array}}{S C_{caller} \vdash expr_b \text{ mem type}^* \text{ arg}^* : sig C_{callee''}}$$

$$\frac{\begin{array}{l} S C_{caller} \vdash expr_b : C_{callee'} \text{ (vref path)} \\ \text{oparser } vis_{glob} \text{ env}_{obj} \text{ func} = \text{find}_{obj}(S, path) \\ C_{callee} = \text{new}[\text{.genv} = C_{caller}.\text{genv}][\text{.oenv} = \text{env}_{obj}] \quad C_{callee'} = \text{res\_glob } C_{callee} \text{ by } vis_{glob} \\ S C_{caller} C_{callee'} \vdash func \text{ type}^* \text{ arg}^* : sig C_{callee''} \end{array}}{S C_{caller} \vdash expr_b \text{ “apply” type}^* \text{ arg}^* : sig C_{callee''}}$$

$$\frac{\begin{array}{l} S C_{caller} \vdash expr_b : C_{callee'} \text{ (vref path)} \\ \text{ocontrol } vis_{glob} \text{ env}_{obj} \text{ func} = \text{find}_{obj}(S, path) \\ C_{callee} = \text{new}[\text{.genv} = C_{caller}.\text{genv}][\text{.oenv} = \text{env}_{obj}] \quad C_{callee'} = \text{res\_glob } C_{callee} \text{ by } vis_{glob} \\ S C_{caller} C_{callee'} \vdash func \text{ type}^* \text{ arg}^* : sig C_{callee''} \end{array}}{S C_{caller} \vdash expr_b \text{ “apply” type}^* \text{ arg}^* : sig C_{callee''}}$$

$$\frac{\begin{array}{l} S C_{caller} \vdash expr_b : C_{callee'} \text{ (vref path)} \\ \text{otable key}^* \text{ action}^* \text{ entry}^* \text{ default custom}^* \text{ func} = \text{find}_{obj}(S, path) \\ S C_{caller} \vdash func \in \in : sig C_{callee''} \end{array}}{S C_{caller} \vdash expr_b \text{ “apply” } \in \in : sig C_{callee''}}$$

### 4.3.2 Restricting the Callee Context Visibility and Passing Control to the Callee

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.

$$\begin{aligned} copyin_{loc'}(C, id \ dir \ type \ expr, val) &= C' && \text{if } dir = out \\ &&& \wedge C \vdash type : rtype \\ &&& \wedge val' = default(rtype) \\ &&& \wedge C' = add_{var_{loc'}}(C, id, rtype, val') \\ copyin_{loc'}(C, id \ dir \ type \ expr, val) &= C' && \text{if } dir = no \vee dir = in \vee dir = inout \\ &&& \wedge C \vdash type : rtype \\ &&& \wedge C' = add_{var_{loc'}}(C, id, rtype, val) \end{aligned}$$

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

$$\begin{aligned} copyin_{loc}(C, \epsilon, \epsilon) &= C \\ copyin_{loc}(C, param \ param^*, val \ val^*) &= C'' && \text{if } C' = copyin_{loc'}(C, param, val) \\ &&& \wedge C'' = copyin_{loc}(C', param^*, val^*) \end{aligned}$$

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

$$\begin{aligned} copyin_{obj'}(C, id \ dir \ type \ expr, val) &= C' && \text{if } dir = out \\ &&& \wedge C \vdash type : rtype \\ &&& \wedge val' = default(rtype) \\ &&& \wedge C' = add_{var_{obj'}}(C, id, rtype, val') \\ copyin_{obj'}(C, id \ dir \ type \ expr, val) &= C' && \text{if } dir = no \vee dir = in \vee dir = inout \\ &&& \wedge C \vdash type : rtype \\ &&& \wedge C' = add_{var_{obj'}}(C, id, rtype, val) \\ copyin_{obj}(C, \epsilon, \epsilon) &= C \\ copyin_{obj}(C, param \ param^*, val \ val^*) &= C'' && \text{if } C' = copyin_{obj'}(C, param, val) \\ &&& \wedge C'' = copyin_{obj}(C', param^*, val^*) \end{aligned}$$

Relation  $copyouts$  defines the copy-out operation.

$$\begin{aligned} [COPYOUT-DO] \ S \ C_{caller} \ C_{callee} \ param \ expr &\hookrightarrow C_{caller'} && \text{if } param = id \ dir \ type \ expr' \\ &&& \wedge dir = inout \vee dir = out \\ &&& \wedge val = find_{var}(C_{callee}, id) \\ &&& \wedge S \ C_{caller} \vdash expr \in val \vdash C_{caller'} \\ [COPYOUT-PASS] \ S \ C_{caller} \ C_{callee} \ param \ expr &\hookrightarrow C_{caller} && \text{if } param = id \ dir \ type \ expr' \\ &&& \wedge dir = no \vee dir = in \\ [COPYOUTS-BASE] \ S \ C_{caller} \ C_{callee} \ \epsilon \ \epsilon &\hookrightarrow C_{caller} \\ [COPYOUTS-REC] \ S \ C_{caller} \ C_{callee} \ (param \ param^*) \ (expr \ expr^*) &\hookrightarrow C_{caller''} && \text{if } S \ C_{caller} \ C_{callee} \ param \ expr \hookrightarrow C_{caller'} \\ &&& \wedge S \ C_{caller'} \ C_{callee} \ param^* \ expr^* \hookrightarrow C_{caller''} \end{aligned}$$

## Intra-object Calls

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

$$\begin{array}{c}
 C_{\text{callee}} = \text{res\_obj } C_{\text{callee}} \text{ by } vis \quad C_{\text{callee}}'' = \text{enter } C_{\text{callee}}' \\
 (param^*, expr^*) = \text{align}_{args}(param^*, arg^*) \\
 S \ C_{\text{callee}} \vdash expr^* : C_{\text{callee}}' \ val^* \\
 C_{\text{callee}}''' = \text{copyin}_{loc}(C_{\text{callee}}'', param^*, val^*) \\
 S \text{ cont } C_{\text{callee}}''' \vdash block : sig \ C_{\text{callee}}''' \\
 S \ C_{\text{callee}}' \ C_{\text{callee}}''' \ param^* \ expr^* \hookrightarrow C_{\text{callee}}'' \\
 C_{\text{callee}}''' = C_{\text{callee}}''[\text{oenv} = C_{\text{callee}}'''\text{.oenv}] \\
 \hline
 S \ C_{\text{callee}} \vdash (\text{faction } vis \ param^* \ block) \ type^* \ arg^* : sig \ C_{\text{callee}}'''
 \end{array}$$

## Inter-object Calls

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

$$\begin{array}{c}
 C_{\text{callee}}' = \text{res\_glob } C_{\text{callee}} \text{ by } vis_{glob} \quad C_{\text{callee}}'' = \text{enter } C_{\text{callee}}' \\
 (param^*, expr^*) = \text{align}_{args}(param^*, arg^*) \\
 S \ C_{\text{callee}} \vdash expr^* : C_{\text{callee}}' \ val^* \\
 C_{\text{callee}}''' = \text{copyin}_{loc}(C_{\text{callee}}'', param^*, val^*) \\
 S \ C_{\text{callee}}''' \vdash id : sig \ C_{\text{callee}}''' \\
 S \ C_{\text{callee}}' \ C_{\text{callee}}''' \ param^* \ expr^* \hookrightarrow C_{\text{callee}}'' \\
 \hline
 S \ C_{\text{callee}} \ C_{\text{callee}} \vdash (\text{fextern } id \ vis_{glob} \ tparam^* \ param^*) \ type^* \ arg^* : sig \ C_{\text{callee}}''
 \end{array}$$

$$\begin{array}{c}
 C_{\text{callee}}' = \text{res\_obj } C_{\text{callee}} \text{ by } vis_{obj} \quad C_{\text{callee}}'' = \text{enter } C_{\text{callee}}' \\
 (param^*, expr^*) = \text{align}_{args}(param^*, arg^*) \\
 S \ C_{\text{callee}} \vdash expr^* : C_{\text{callee}}' \ val^* \\
 C_{\text{callee}}''' = \text{copyin}_{obj}(C_{\text{callee}}'', param^*, val^*) \\
 S \ C_{\text{callee}}''' \vdash id : sig \ C_{\text{callee}}''' \\
 S \ C_{\text{callee}}' \ C_{\text{callee}}''' \ param^* \ expr^* \hookrightarrow C_{\text{callee}}'' \\
 \hline
 S \ C_{\text{callee}} \ C_{\text{callee}} \vdash (\text{fexternmethod } id \ vis_{obj} \ tparam^* \ param^*) \ type^* \ arg^* : sig \ C_{\text{callee}}''
 \end{array}$$

$$\begin{array}{c}
 C_{\text{callee}}' = \text{res\_obj } C_{\text{callee}} \text{ by } vis_{obj} \\
 (param^*, expr^*) = \text{align}_{args}(param^*, arg^*) \\
 S \ C_{\text{callee}} \vdash expr^* : C_{\text{callee}}' \ val^* \\
 C_{\text{callee}}'' = \text{copyin}_{obj}(C_{\text{callee}}', param^*, val^*) \\
 S \text{ cont } C_{\text{callee}}'' \vdash block : sig \ C_{\text{callee}}'' \\
 S \ C_{\text{callee}}' \ C_{\text{callee}}'' \ param^* \ expr^* \hookrightarrow C_{\text{callee}}'' \\
 \hline
 S \ C_{\text{callee}} \ C_{\text{callee}} \vdash (\text{fmethod } vis_{obj} \ tparam^* \ param^* \ block) \ type^* \ arg^* : sig \ C_{\text{callee}}''
 \end{array}$$

$$\begin{array}{c}
 C_{\text{callee}} = \text{res\_glob } C_{\text{callee}} \text{ by } vis \quad C_{\text{callee}}'' = \text{enter } C_{\text{callee}}' \\
 (param^*, expr^*) = \text{align}_{args}(param^*, arg^*) \\
 S \ C_{\text{callee}} \vdash expr^* : C_{\text{callee}}' \ val^* \\
 C_{\text{callee}}''' = \text{copyin}_{loc}(C_{\text{callee}}'', param^*, val^*) \\
 S \text{ cont } C_{\text{callee}}''' \vdash block : sig \ C_{\text{callee}}''' \\
 S \ C_{\text{callee}}' \ C_{\text{callee}}''' \ param^* \ expr^* \hookrightarrow C_{\text{callee}}'' \\
 \hline
 S \ C_{\text{callee}} \ C_{\text{callee}} \vdash (\text{faction } vis \ param^* \ block) \ type^* \ arg^* : sig \ C_{\text{callee}}''
 \end{array}$$