P4Cub: A Little Language for Big Routers

Rudy Peterson
Cornell University
Ithaca, NY, USA
rnp39@cornell.edu

Eric Hayden Campbell
Cornell University
Ithaca, NY, USA
ehc86@cornell.edu

John Chen
Cornell University
Ithaca, NY, USA
jc2786@cornell.edu

Natalie Isak
Microsoft
Cambridge, MA, USA
ngi2@cornell.edu

Calvin Shyu
Cornell University
Ithaca, NY, USA
cws225@cornell.edu

Ryan Doenges
Cornell University
Ithaca, NY, USA
rhd89@cornell.edu

Parisa Ataei
Cornell University
Ithaca, NY, USA
psa43@cornell.edu

Nate Foster
Cornell University
Ithaca, NY, USA
jnfoster@cs.cornell.edu

Abstract

P4Cub is a new intermediate representation (IR) for the P4 programming language that is designed to facilitate the development of certified tools. It is organized around a small set of core constructs that avoid complexities found in the surface language such as side effects in expressions, mutual recursion between the expressions and statements, and so on. Still, P4Cub retains the essential domain-specific features of P4 itself. P4Cub has a front-end based on Petr4, and has been fully mechanized in Coq including big-step and small-step semantics and a type system. We have built several certified tools using P4Cub including a type soundness proof, a compiler pass, and an automated verification tool.

Keywords: Coq, P4, formal semantics, formal verification, intermediate representations, domain-specific languages.

ACM Reference Format:
Rudy Peterson, Eric Hayden Campbell, John Chen, Natalie Isak, Calvin Shyu, Ryan Doenges, Parisa Ataei, and Nate Foster. 2023. P4Cub: A Little Language for Big Routers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3573105.3575670

CPP '23, January 16–17, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0026-2/23/01…$15.00
https://doi.org/10.1145/3573105.3575670

1 Introduction

Well-designed intermediate representations (IR) underpin some of the most successful compiler frameworks including LLVM [Lattner and Adve 2004] and CompCert [Leroy et al. 2016]. IRs enforce abstraction boundaries between source and target languages and they also influence the design of compiler passes that translate between them. In the context of mechanized compilers like CompCert, IRs affect the structure and complexity of correctness proofs. However, existing mechanized IRs are ill-suited for reasoning about domain-specific languages because they are based on general-purpose programming constructs.

This paper presents a new mechanized IR for P4 called P4Cub. P4 is a domain-specific language for network data planes that is seeing growing use both as a language for specifying functionality on programmable devices (switches, NICs, end-hosts, etc.) and as a language for modeling the behavior of conventional, fixed-function devices (e.g., Google uses P4 to model their data center switches for differential testing [Albab et al. 2022]).

Existing formalizations of P4 are based on the language's surface syntax, which is complex and unwieldy to work with [Doenges et al. 2021]. Where a P4 programmer sees flexible syntax and expressive abstractions, proof engineers see convoluted semantics and knotty inductive proofs. Of course, similar challenges arise in other languages, but they are particularly egregious in the case of P4, as the language has very little essential complexity. Fortunately, as it turns out, there is an elegant language embedded within P4—it just needs to be pulled out into a "little language" of its own.

At a high level, our design for P4Cub is based on two main considerations. First, we exploit P4's essential simplicity—it has no loops, recursion, memory management, dynamic allocation, or higher-order features—to design a core language organized around a set of simple and orthogonal constructs. We demonstrate how to compile P4's surface language into

P4Cub, and we highlight how our static and dynamic semantics eliminate redundant rules.

Second, we embed P4Cub into Coq in a manner that seeks to streamline the development of formal proofs. For example, although P4's surface syntax is presented using named variables, P4Cub uses a nameless representation of terms. As has been shown in prior work, nameless representations can simplify mechanized proofs, since α-equivalence comes for free. Similarly, while P4 allows side effects like function calls and match-action table invocations to appear in both expressions and statements, P4Cub requires all side effects to occur at the statement level, which eliminates a tricky mutual recursion between the two. We provide a compiler pass to lift all side effects occurring in expressions up to the statement level.

At the same time, P4Cub does not distill P4 down to its absolute essence. Instead, it strives to retain the central features of P4 such as header types, parsers, and match-action tables. This approach allows P4 experts to carry out proofs in terms of familiar, relatively high-level, domain-specific constructs. As we show using case studies, P4Cub can be readily applied to a variety of problems including proofs of type soundness, verification of compilers, and construction of tools for verifying P4 programs themselves.

The rest of the paper is organized as follows. First, we give a brief overview to P4 and P4Cub (Section 2). Next, we define P4Cub's syntax (Section 3) and semantics (Section 4). After that, we present our Coq implementation (Section 5) and case studies (Section 6). Finally, we discuss related work (Section 7) and conclude with a brief discussion of possible directions for future work (Section 8).

2 Overview

P4 is a domain-specific language based on a collection of relatively high-level abstractions for specifying network data planes. The core of P4 is based on a relatively simple imperative language, extended with a few domain-specific constructs such as header types, parser state machines, and match-action tables. We briefly review these constructs for readers unfamiliar with the language, before highlighting a few representative aspects of our design for P4Cub.

P4's *header types* and *parser state machines* convert packets into typed representations that can be manipulated in the rest of the program.

```
header ethernet_t { bit<48> dstAddr;
                    bit<48> srcAddr;
                    bit<16> ethTyp;  }
parser MyParser(packet_in packet,
    out headers hdr, inout metadata meta,
    inout standard_metadata_t standard_metadata) {
  state start {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.ethTyp) {
      0x8100: reject;
      default: accept;
    } } }
```

In this example, the header type captures the standard format for Ethernet packets with 112 bits. The parser extracts 112 bits from the packet and performs a simple form of validation, checking that the Ethernet type field is not `0x8100` (i.e., that the packet does not carry a VLAN tag).

P4's *match-action tables* describe configurable procedures that can be managed by the control-plane at runtime—either a traditional distributed routing protocol or a software-defined networking controller.

```
control MyIngress(inout headers hdr, inout metadata meta,
    inout standard_metadata_t standard_metadata) {
  action drop() {
    mark_to_drop(standard_metadata);
  }
  action fwd(bit<9> port) {
    standard_metadata.egress_spec = port;
  }
  table sw {
    key = { hdr.ethernet.dstAddr: exact; }
    actions = { fwd; drop; }
  }
  apply { sw.apply(); }
}
```

Here, the `control` block consists of a single match-action table `sw` that looks up the destination address in the Ethernet header in the table and either forwards the packet or drops it. Note that the semantics of the table is not specified by the P4 program itself—to understand whether and how it forwards packets, we need to know the values of the keys and actions of the entries in the table.

In addition to these domain-specific features, P4 provides a number of other constructs. As features have been added over time, the language has grown in size and complexity, which makes it harder to build implementations. In the rest of this section, we highlight a few of the complexities that arise in P4's surface syntax, and briefly discuss how they are streamlined in P4Cub.

Example 1. P4's type system provides domain-specific constructs for modeling the structure of packets, as well as standard constructs for organizing other program data, often leading to redundancy. For instance, P4 includes `header` and `struct` types, both of which describe record-like structures whose values can be accessed using "dot" notation. Following is a `struct` that could be used to encode the headers found in a standard TCP/IP packet.

```
struct headers {
  ethernet_t ethernet;
  ipv4_t ipv4;
  tcp_t tcp;
}
```

Despite the differences between `header` and `struct` types—e.g., values of the former type have a validity bit that tracks initialization and the fields are serialized in declaration order, whereas values of the latter do not have a validity bit and have unordered fields—we chose to combine the two into a single type in P4Cub, using a boolean flag to distinguish the

minor differences in their semantics. Similarly, P4's header stacks, which can be used to capture the structure of MPLS packets among others, are encoded in P4Cub using standard arrays, which eliminates another form of redundancy at the type system level.

Example 2. P4's original design lacked functions,[1] but it has always allowed `parser` and `control` declarations to be used as macros, factoring out common functionality into reusable blocks of code that can be instantiated many times. For instance, the `control` declaration below models a generic access-control table that forwards or drops the packet based on a single byte:

```
control acl(inout bit<8> k)() {
  table t {
    key = { k : exact }
    actions = { drop; forward }
  }
  apply { t.apply() }
}
```

This `control` can be instantiated and invoked multiple times in the "main" `control` on different arguments:

```
control c(...) {
  acl() c1;
  acl() c2;
  apply {
    c1.apply(x);
    c2.apply(y);
  }
}
```

P4 imposes restrictions to ensure that a `control` used in this way can always be flattened and inlined into a single top-level `control`:

```
control c(...) {
  table t1 {
    key = { x : exact; }
    action = { drop; forward; }
  }
  table t2 {
    key = { y : exact; }
    action = { drop; forward; }
  }
  apply {
    t1.apply();
    t2.apply();
  }
}
```

In contrast, P4Cub disallows nested `parser` and `control` instantiations and instead requires them to be instantiated at the top level—nested instantiations do not increase the expressiveness of the language, and they can always be inlined as in the example.

Example 3. As a final example, P4 allows match-action tables to be invoked from expressions, and also supports branching on the results of table invocation:

```
switch (sw.apply().action_run) {
  fwd: { f.count(); }
  drop: { r.count(); }
}
```

For simplicity, P4Cub only allows table invocations at the statement level, and requires branching on the results to be implemented using standard conditionals. The front-end provides translations to convert programs written in the surface syntax into IR programs that satisfy these restrictions.

3 Syntax

With this background, we are now ready to introduce P4Cub itself. The syntax of P4Cub has many elements of a standard imperative language, including arithmetic, structs, arrays, and assignment. It also retains the domain-specific features of P4 meant to reflect common idioms found in network programs, even though it would be possible to encode them in terms of other constructs—e.g., tables could become conditionals. This design choice ensures that programs can be configured by the control-plane and readily compiled to a variety of targets.

As discussed in the introduction, the primary goal of our design for P4Cub is to streamline formal, mechanized reasoning about P4 programs. Toward this goal, P4Cub's syntax is based on three primary ideas. First, we eliminate many P4 features including strings, enums, header-unions (C-style unions of header-types in P4), and header-stacks (arrays of headers) by compiling them into simpler constructs. Second, we adopt de Bruijn indices for type and term variables in the mechanization, to ease reasoning about compiler transformations, especially ones that introduce new variable declarations. Third, we limit side effects to statements. In other words, similar to Clight, side effects may not be arbitrarily nested deep in expressions. Instead, they must appear at the statement level. This restriction makes P4Cub's semantics simpler and eliminates mutual induction between statements and expressions in proofs.

Formally, P4Cub's syntax is divided into types, expressions, parser-transition expressions, statements, declarations within controls, and top-level declarations, as shown in Figure 1 through Figure 3 (see the appendix for declarations). A reference to the metavariables used throughout the paper is provided in Table 1.

P4Cub expression types, shown in Figure 1, include base types such as bit-strings $\mathsf{bit}\langle n \rangle$, type variables, arrays, headers, and structs. Type variables are encoded with de Bruijn indices. Conceptually, P4 headers are struct-like datatypes that represent packet headers in the networking sense, e.g., an IP header or an Ethernet header. These headers are segmented into fields which specify addresses, flags, and the like–data that often varies in size and may not even be byte-aligned. To accommodate this, numeric datatypes in P4 have the form $\mathsf{bit}\langle n \rangle$ and $\mathsf{int}\langle p \rangle$, with widths of n and p bits respectively. Unlike P4, widths do not need to be multiples of

[1] Top-level functions were added to the language in version 1.1.0, but with a number of restrictions [P4 Language Consortium 2022].

Table 1. Metavariables.

Symbol	Name	Symbol	Name
τ	type	b	bool
z	integer	n	natural number
p	positive number	x	string
arg	argument	prm	parameter
e	expression	pat	select pattern
l	parser state label	pt	parser transition
s	statement	cd	control declaration
td	top declaration	v	value
lv	left-value	arg_v	evaluated argument
ctx	syntactic context	Γ	typing environment
ϵ	value environment	sig	typing signals
signal	evaluation signals	ψ	extern state
fns_t	function types	fns	functions
$inst_t$	instance type	$insts_t$	instance types
$inst$	instance	$insts$	instances

8. Since headers are similar to structs we represent both by $\text{struct}_b\ \overline{\tau}$, where b is `true` for headers and `false` for standard structs and $\overline{\tau}$ is a list of types that corresponds to fields since field names are natural numbers instead of identifiers. It is important to distinguish the two because there are some differences between `header` and `struct` types—e.g., values of the former type have a validity bit that tracks initialization and the fields are serialized in declaration order, whereas the latter do not have a validity bit and have unordered fields.

Example 4. The following code snippet shows the P4Cub encoding of the *headers* struct and *ethernet_t* header, which were used in Example 1.

```
struct false {
    struct true {
      bit<48> ;
      bit<48> ;
      bit<16> ; } ;
    ...
}
```

Note that fields do not have names and that type declarations must be inlined. P4Cub also requires type synonyms and constants to be inlined. For better optimization, P4Cub flattens declarations and hoists instantiations to the top level.

P4Cub expressions, shown in Figure 1, share primitive P4 operations such as bit-slicing, casts, arithmetic, and struct membership. Term variables also use de Bruijn indices. List literals including structs, headers, and arrays are collapsed into one Coq constructor. Squishing multiple constructs into one reduces case analyses in proofs—it prevents having to prove similar cases for all three variants. Structs and headers are accessed by a natural number whereas arrays are indexed by an arbitrary numeric expression, modulo restrictions set by the type system. Like P4, P4Cub separates parser transition expressions from expressions and distinguishes declarations within controls from top-level declarations.

Types:

τ	::=	bool	*booleans*
	|	$\text{bit}\langle n\rangle$	*unsigned integers*
	|	$\text{int}\langle p\rangle$	*signed integers*
	|	$\tau[n]$	*arrays*
	|	$\text{struct}_b\ \overline{\tau}$	*structs/headers*
	|	n	*type variables*

Operators:

$\ominus$	::=	! | ~ | −
$\oplus$	::=	+ | − | ∗ | ÷ | mod
	|	== | != | && | || | |+| | |−|
	|	& | | | ^ | ~ | ++ | ≪ | ≫
	|	< | ≤ | > | ≥

Expressions:

e	::=	b	*boolean*
	|	$z\langle n\rangle$	*unsigned integer*
	|	$z\langle p\rangle$	*signed integer*
	|	$\tau\ n$	*variable*
	|	$e[p:p]$	*bit-slicing*
	|	$(\tau)\ e$	*cast*
	|	$\ominus\ e$	*unary operation*
	|	$e \oplus e$	*binary operation*
	|	$\{\overline{e}\}$	*list literal*
	|	$e[e]$	*array indexing*
	|	$e.n$	*struct member*

Select Patterns:

pat	::=	_	*wild pattern*
	|	$z\langle n\rangle$	*unsigned integer*
	|	$z\langle p\rangle$	*signed integer*
	|	pat &&& pat	*bit-mask*
	|	pat .. pat	*range*
	|	$\overline{pat}$	*list/struct pattern*

Parser State Labels:

l	::=	start	*start label*
	|	accept	*accept label*
	|	reject	*reject label*
	|	n	*user-defined label*

Parser Transition Expressions:

pt	::=	direct l	*direct state transition*
	|	select $e\ l\ \{\overline{pat \Rightarrow l}\}$	*select transition*

Figure 1. P4Cub expression syntax.

Arguments and parameters do not have names, defined in Figure 2, as they use de Bruijn indices. There are three kinds of parameters:

Arguments:

arg	::=	in e	*in-arguments*
	|	out e	*out-arguments*
	|	inout e	*inout-arguments*

Parameters:

prm	::=	in τ	*in-parameters*
	|	out τ	*out-parameters*
	|	inout τ	*inout-parameters*

Figure 2. P4Cub arguments and parameters.

Statements:

s	::=	skip	*skip*
	|	return e	*return*
	|	exit	*exit*
	|	goto *pt*	*parser transition*
	|	$e := e$	*assignment*
	|	$e\ x\langle\overline{\tau}\rangle(\overline{arg})$	*function call*
	|	$x\ (\overline{e}, \overline{arg})$	*action call*
	|	$e\ x\ x\langle\overline{\tau}\rangle(\overline{arg})$	*method call*
	|	invoke x	*table invocation*
	|	apply $x(\overline{arg})$	*apply statements*
	|	let e in s	*let binding*
	|	s; s	*sequencing*
	|	if e then s else s	*conditional*

Figure 3. P4Cub statement syntax.

- *in* parameters are read-only and are initialized by copying the value of the corresponding argument when the invocation is executed;
- *out* parameters are uninitialized; an argument passed as an *out* parameter must be accompanied with a storage reference (an l-value), and after the execution of the call, the value of the parameter is copied to the corresponding storage location; and
- *inout* parameters are both in and out.

Statements in P4Cub, shown in Figure 3, can be divided into atomic statements, such as skip, return e, parser transition statements; and compound statements that determine the program's control flow, such as conditionals and sequencing. Atomic statements end statement blocks and do not introduce new variables (de Bruijn identifiers) into scope. Variable declarations let e in s shift the de Bruijn context up by binding e to de Bruijn index 0 in block s, thus, it does not escape the scope of s. P4Cub only allows side effects at the statement level. Thus, function calls, invocation of tables, applications of parsers and controls, and extern method calls must be statements. For instance, the P4 code shown in Example 3 would be written as Example 5 in P4Cub.

Example 5. Note that since `action_run` is the third field of the `apply_result` `struct` for `table` `sw` it has been transformed to field 2. The invoke must occur at the statement level, no longer embedded in the field projection. Furthermore the `enum` members for `action_run` are compiled to unsigned integers, where the width represents the number of members and the value the position in the member list. There are no switch statements in P4Cub so it becomes a nested conditional where each guard checks equality to a member of the enum.

```
var sw.invoke();
if 0.2 = 2W0 {
  f.count();
} else if 0.2 = 2W1 {
  r.count();
} else {
  skip
}
```

Like P4, P4Cub distinguishes between different kinds of procedure calls. P4Cub programs can call functions, actions, tables, external methods, parsers, and controls. Each kind of call behaves differently and represents a different component of a packet-processing pipeline.

Example 6. We show the `MyIngress` control illustrated in Section 2 in P4Cub.

```
control MyIngress()
(out struct false { header true  { bit<48> ; bit<48> ;
     bit<16> }  },
inout struct false { },
inout struct false { ... }) {
  action drop()() { mark_to_drop(2); }
  action fwd(bit<9>)() { 3.1 = 0; }
  table sw {
    key = { 0.0.0: exact; }
    actions = { fwd; drop; } }
  apply { sw.invoke; }
}
```

The code for `fwd` illustrates the de Bruijn indices in play. The input structs are respectively `headers`, `metadata`, and `standard_metadata_t`. The original variable for the latter has de Bruijn index 2 which becomes 3 because of `fwd`'s argument. Controls also have separate extern arguments.

4 Static and Dynamic Semantics

Figure 4 defines the environments, stores, and contexts used in P4Cub's type system. Figure 5 defines such for P4Cub's operational semantics. Environments Γ and stores ϵ are lists of types and values, respectively, that associate de Bruijn indices (the list's indices) to types or values, respectively. Thus, looking up a variable n's type from the environment Γ returns the type at the n's index in the environment, denoted $\Gamma\ n$. The same notation is used to look up a variable in the store. We write $\tau :: \Gamma$ to indicate appending τ to the "beginning" of the environment.

The function fns_t maps a function's name to the number of type parameters, expression parameters, and return type.

Expression Typing Environment:

Γ	::=	$\overline{\tau}$	*A list of types*

Typing Function Environment:

fns_t	::=	$\emptyset$	*Empty*
	|	$fns_t, x \mapsto (n, \overline{\tau}, \tau)$	*Signature*

Instance Types:

$inst_t$	::=	$\overline{\tau}, \overline{\tau}$	*Action signature*
	|	$\overline{(x, n, \overline{\tau}, \tau)}$	*Extern type*
	|	Prsr $\overline{\tau}$	*Parser type*
	|	Ctrl $\overline{\tau}$	*Control type*
	|	Table	*Table*

Typing Environment:

$insts_t$	::=	$\emptyset$	
	|	$insts_t, x \mapsto inst_t$	

Typing Syntactic Contexts:

ctx	::=	Prsr n $insts_t$	*Parser*
	|	Ctrl $insts_t$	*Control*
	|	Fn τ	*Function*

Typing Signal:

sig	::=	Cont	*continue*
	|	Exit	*exit*
	|	Return τ	*return a type*
	|	Trans	*transition*

Figure 4. Typing environment, context, and signal syntax.

Expression Evaluation Store:

ϵ	::=	$\overline{v}$	*A list of values*

Evaluation Function Environment:

fns	::=	$\emptyset$	*Empty*
	|	$fns, x \mapsto (fns, s)$	*Closure*

Evaluation Instances:

$inst$	::=	$\epsilon, insts, s$	*Action closure*
	|	$fns, insts, s$	*Control*
	|	$fns, insts, s, \overline{s}$	*Parser*
	|	$n, \overline{e}, \overline{(x, \overline{arg})}$	*Table*

Evaluation Environment:

$insts$	::=	$\emptyset$	
	|	$insts, x \mapsto inst$	

Evaluation Syntactic Contexts:

ctx	::=	Prsr n s $\overline{s}$ $insts$	*Parser*
	|	Ctrl $insts$	*Control*
	|	Function	*Function*

Evaluation Signal:

signal	::=	Cont	*continue*
	|	Exit	*exit*
	|	Return v	*return a value*
	|	Accept	*accept*
	|	Reject	*reject*

Figure 5. Eval. environment, context, and signal syntax.

P4Cub stores instance information—such as a parser instance or a control instance—in an "instance" type and supports instance types for actions, externs, parsers, controls, and tables. Action types contain the signature of control-plane parameters and that of data-plane parameters. Extern instance types include the name of each method with its signature, much like a function's signature. Parser and control instances each contain the types of runtime parameters. Tables do not need a signature as they are only invoked with their name. In the paper all instance types are kept in one environment $insts_t$ which maps names to instance types. In the implementation they are kept in separate namespaces.

The function fns maps a function's name to the available functions in scope and its body. The instance types $inst_t$ have corresponding instances $inst$ for evaluation. Action closures have the local expression and instance environment and the action's body. Control instances include the local function and instance environment and the control's apply block. Parser instances also include the local function and instance environment as well as the parser's start and user-defined states. Table declarations are paired with the number of term variables declared in the control before it. As will be shown in the evaluation rule for tables, this is used to split the store to evaluate the match-action table. Instances for externs are not included here because they are handled internally by the target-dependent extern environment ψ. Again for expository simplicity, all "instances" are in the same environment but in the implementation have separate namespaces.

The typing syntactic context ctx defines the syntactic context where a statement is placed and it contains different scope information of the statement for each kind of context. For example, the Ctrl $insts_t$ is used when inside a control declaration, such as an action declaration or a control's apply block. It has information such as the tables defined within the current control, the actions declared, and other control instances in scope. This information is not needed when typing or evaluating a parser state. The Prsr n $insts_t$ contains information only needed for parsers, such as the number of states of the current parser-state machine and other parser instances in scope. These two contexts also contain available extern type signatures. The Fn τ is used when inside of a top-level defined function with the return type τ. Unlike other contexts, it does not provide any information about

parser, control, nor extern instances because functions in P4 are not allowed to invoke any of these.

Evaluation syntactic contexts are used in the dynamic semantics to provide environments and information about the enclosing syntactic context. Similarly to the typing syntactic contexts the evaluation version includes information particular to different blocks of a program. The parser context includes the number of parameters, start state, and user-defined states of the enclosing parser, as well as the available parser instances in scope. The control context includes the tables and actions of the enclosing control, as well as the available control instances in scope.

A typing or evaluation signal, sig and signal, respectively, indicates whether control flow continues. They are also used to check that a statement is properly formed within its context. Signals such as Cont and Return are essentially the same as in other imperative languages. An Exit signal indicates that the entire program should stop evaluating. Indeed, Exit halts execution all the way up to the packet-processing pipeline level, whereas Return only interrupts the enclosing statement block. The Return τ typing signal returns a type while the evaluation Return v signal returns a value. Trans is similar to return but for the parser-state machine and it helps to verify that a parser-state terminates with a transition statement. In our implementation, more specific signals are used to embody if the packet was accepted or rejected in parsing: Accept and Reject.

4.1 Type System

Figure 6 shows the expression typing rules, most of which are straightforward. As just mentioned, the environment Γ is a list of types where the index of a type is de Bruijn term identifier and $\Gamma\ n$ denotes looking up the n^{th} variable in the environment. We use the same notation for look up in any list. For example, the T-Member rule states that the n's member of expression e has the type τ if expression e has a struct type where its n's field has the type τ, which is denoted by the look up function $\overline{\tau}\ n$. In T-BinOp, the helper function bop_type $\oplus\ \tau_1\ \tau_2$ determines the type of the expression based on the binary operator and its operands. As an example, bop_type + bit$\langle n \rangle$ bit$\langle n \rangle$ = bit$\langle n \rangle$. We also take advantage of P4's numeric data types such as bit$\langle n \rangle$, which permit one to specify unsigned integers bound by 2^n. As an example, T-Index allows any term of type bit$\langle n \rangle$ to index into an array, because the length of the array is the upper-bound on values of such terms. This ensures that evaluating a well-typed array index expression cannot cause an out of bounds error. Typing a list expression $\{\overline{e}\}$ just types its elements, that is, $\Gamma \vdash \overline{e} : \overline{\tau}$.

Figure 6 also shows statement typing rules. Note that by using de Bruijn indices we eliminate the need to update the environment—new variables are only introduced in a local scope by let e in s. Thus, the T-LetIn is the only place where the environment is locally extended. Additionally, no bindings "leak," so there is no need to produce an environment with the declared variable bound.

Terminal statements such as exits, returns, and transitions produce a unique signal. Some rules such as T-ActCall, T-ApplyCtrl, and T-Invoke look up signatures of the invokee in the syntactic context rather than Γ. This is due to the fact that some P4 constructs can only be called in certain places which is captured by the context. For instance, transitioning to a different parser state, shown in T-Transition rule, is only reasonable in a parser context.

The statement typing rules use multiple helper judgments, provided in Appendix A, which use a subscript under their inference symbol—e.g., $n, \Gamma \vdash_p pt$ represents the judgment form of parser transition typing. The rules use some helper functions and predicates. The T-FunCall rule states that calling the function x with the return expression e_r, type arguments $\overline{\tau_{arg}}$, and arguments $\overline{arg}$ results in a Cont signal if x exists in the context and it has $|\overline{\tau_{arg}}|$ type parameters, the return type τ_r, and parameters $\overline{\tau}$; the return expression e_r can be evaluated to an l-value (denoted by helper predicate lvalue_ok e_r), and e_r and $\overline{arg}$ type check. To type check e_r and $\overline{arg}$ type substitutions must be performed using the type arguments $\overline{\tau_{\text{arg}}}$. e_r is typed as return type τ_r substituted with type arguments $\overline{\tau_{\text{arg}}}$ (denoted by the helper function tsub $\overline{\tau_{\text{arg}}}\ \tau_r$). The arguments $\overline{arg}$ are typed as the parameters $\overline{\tau}$ substituted with type arguments $\overline{\tau_{\text{arg}}}$ (again, denoted by the helper tsub $\overline{\tau_{\text{arg}}}\ \overline{\tau}$). tsub $\overline{\tau}\ \tau$ substitutes de Bruijn type variables in τ with $\overline{\tau}$: the first type argument is substituted for 0, the second for 1, and so on. Note that we take advantage of the list notation when a judgment or function is being mapped to a list. For example, in T-ApplyCtrl, $\Gamma \vdash_{arg} \overline{arg} : \overline{\tau}$ states that arguments $\overline{arg}$ have the type $\overline{\tau}$.

4.2 Evaluation

Figure 7 shows the big-step semantics of expressions. Expressions evaluate to values, defined in Figure 8. Additionally, sometimes expressions are partially evaluated to *l-values*, also defined in Figure 8. L-values represent assignable locations, such as an array index, a struct field, or a variable.

De Bruijn stores ϵ are a list of values. Expressions do not introduce new variables so no de Bruijn shifts are required. The rules are self-explanatory. Similar to typing of a list expression, the evaluation of it is also just mapping the judgment onto the list, that is, $\langle \epsilon, \overline{e} \rangle \Downarrow \overline{v}$.

Figure 9 shows the big-step semantics of statements. For evaluating statements we need a store ϵ, a context ctx, and an extern state ψ. ψ represents the state of *external* objects (also known as `externs`). P4Cub takes advantage of direct Coq definitions of targets and externs. Thus, in our formalization here, we leave the definition mostly opaque. This detail is hidden in helper functions such as exec_extern. For instance, E-MtdCall is the only rule that changes the extern

P4Cub's expression typing rules:

$$\boxed{\Gamma \vdash e : \tau} \qquad \frac{}{\Gamma \vdash b : \text{bool}}\ \text{T-Bool} \qquad \frac{0 \le z < 2^n}{\Gamma \vdash z\langle n \rangle : \text{bit}\langle n \rangle}\ \text{T-Bit} \qquad \frac{-2^{p-1} \le z < 2^{p-1}}{\Gamma \vdash z\langle p \rangle : \text{int}\langle p \rangle}\ \text{T-Int}$$

$$\frac{\Gamma\ n = \tau}{\Gamma \vdash \tau\ n : \tau}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \text{struct}_b\ \overline{\tau} \qquad \overline{\tau}\ n = \tau}{\Gamma \vdash e.n : \tau}\ \text{T-Mem} \qquad \frac{\Gamma \vdash e : \tau \qquad \text{numeric\_width}\ n\ \tau \qquad p_1 \le p_2 < n}{\Gamma \vdash e[p_2 : p_1] : \text{bit}\langle p_2 - p_1 + 1 \rangle}\ \text{T-Slice}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \text{proper\_cast}\ \tau'\ \tau}{\Gamma \vdash (\tau)\ e : \tau}\ \text{T-Cst} \qquad \frac{\Gamma \vdash e : \tau \qquad \text{uop\_type} \ominus\ \tau\ \tau'}{\Gamma \vdash \ominus e : \tau'}\ \text{T-Un} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \text{bop\_type} \oplus\ \tau_1\ \tau_2 = \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}\ \text{T-Bi}$$

$$\frac{\Gamma \vdash e_1 : \tau[2^n] \qquad \Gamma \vdash e_2 : \text{bit}\langle n \rangle}{\Gamma \vdash e_1[e_2] : \tau}\ \text{T-Idx} \qquad \frac{\overline{\Gamma \vdash e : \tau}}{\Gamma \vdash \{\overline{e}\} : \{\overline{\tau}\}}\ \text{T-Lists}$$

P4Cub's statement typing rules:

$$\boxed{\Gamma, \mathit{fns}_t, \text{ctx} \vdash s \dashv \text{sig}} \qquad \frac{}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash \text{skip} \dashv \text{Cont}}\ \text{T-Skip} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, \mathit{fns}_t, \text{Fn}\ \tau \vdash \text{return}\ e \dashv \text{Return}}\ \text{T-Rtrn}$$

$$\frac{\text{exit\_ok ctx}}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash \text{exit} \dashv \text{Exit}}\ \text{T-Exit} \qquad \frac{n, \Gamma \vdash_p pt}{\Gamma, \mathit{fns}_t, \text{Prsr}\ n\ \mathit{insts}_t \vdash \text{goto}\ pt \dashv \text{Trans}}\ \text{T-Trans}$$

$$\frac{\text{lvalue\_ok}\ e_1 \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash e_1 := e_2 \dashv \text{Cont}}\ \text{T-Asgn} \qquad \frac{\mathit{insts}_t\ x = (\overline{\tau_c}, \overline{\tau_d}) \qquad \Gamma \vdash \overline{e} : \overline{\tau_c} \qquad \Gamma \vdash_{arg} \overline{arg} : \overline{\tau_d}}{\Gamma, \mathit{fns}_t, \text{Ctrl}\ \mathit{insts}_t \vdash x\ (\overline{e}, \overline{arg}) \dashv \text{Cont}}\ \text{T-ActCall}$$

$$\frac{\mathit{fns}_t\ x = |\overline{\tau_{\text{arg}}}|, \overline{\tau}, \tau_r \qquad \text{lvalue\_ok}\ e_r \qquad \Gamma \vdash e_r : \text{tsub}\ \overline{\tau_{\text{arg}}}\ \tau_r \qquad \Gamma \vdash_{arg} \overline{arg} : \text{tsub}\ \overline{\tau_{\text{arg}}}\ \overline{\tau}}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash e_r\ x\langle \overline{\tau_{\text{arg}}} \rangle(\overline{arg}) \dashv \text{Cont}}\ \text{T-FunCall}$$

$$\frac{\text{ctx}\ x\ x_m = (|\overline{\tau_{\text{arg}}}|, \overline{\tau}, \tau_r) \qquad \text{lvalue\_ok}\ e_r \qquad \Gamma \vdash e_r : \text{tsub}\ \overline{\tau_{\text{arg}}}\ \tau_r \qquad \Gamma \vdash_{arg} \overline{arg} : \text{tsub}\ \overline{\tau_{\text{arg}}}\ \overline{\tau}}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash e_r\ x\ x_m\langle \overline{\tau_{\text{arg}}} \rangle(\overline{arg}) \dashv \text{Cont}}\ \text{T-MtdCall}$$

$$\frac{\mathit{insts}_t\ x = \text{Ctrl}\ \overline{\tau} \qquad \Gamma \vdash_{arg} \overline{arg} : \overline{\tau}}{\Gamma, \mathit{fns}_t, \text{Ctrl}\ \mathit{insts}_t \vdash \text{apply}\ x(\overline{arg}) \dashv \text{Cont}}\ \text{T-ApplyCtrl} \qquad \frac{\mathit{insts}_t\ x = \text{Prsr}\ \overline{\tau} \qquad \Gamma \vdash_{arg} \overline{arg} : \overline{\tau}}{\Gamma, \mathit{fns}_t, \text{Prsr}\ n\ \mathit{insts}_t \vdash \text{apply}\ x(\overline{arg}) \dashv \text{Cont}}\ \text{T-ApplyPrsr}$$

$$\frac{\mathit{insts}_t\ x = \text{Table}}{\Gamma, \mathit{fns}_t, \text{Ctrl}\ \mathit{insts}_t \vdash \text{invoke}\ x \dashv \text{Cont}}\ \text{T-Invoke} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau :: \Gamma, \mathit{fns}_t, \text{ctx} \vdash s \dashv \text{sig}}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash \text{let}\ e\ \text{in}\ s \dashv \text{sig}}\ \text{T-LetIn}$$

$$\frac{\Gamma, \mathit{fns}_t, \text{ctx} \vdash s_1 \dashv \text{Cont} \qquad \Gamma, \mathit{fns}_t, \text{ctx} \vdash s_2 \dashv \text{sig}}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash s_1; s_2 \dashv \text{sig}}\ \text{T-Seq} \qquad \frac{\Gamma \vdash e : \text{bool} \qquad \Gamma, \mathit{fns}_t, \text{ctx} \vdash s_1 \dashv \text{sig}_1 \qquad \Gamma, \mathit{fns}_t, \text{ctx} \vdash s_2 \dashv \text{sig}_2}{\Gamma, \mathit{fns}_t, \text{ctx} \vdash \text{if}\ e\ \text{then}\ s_1\ \text{else}\ s_2 \dashv \text{lub sig}_1\ \text{sig}_2}\ \text{T-Condi}$$

Figure 6. P4Cub expression and statement typing.

environment ψ, all other statement evaluation rules simply propagate such a change.

The left-hand side of an assignment in the E-Assign rule and some arguments in function calls in the E-FunCall rule are partially evaluated to l-values. This is because we want to get a *location* they represent in the environment ϵ that can be used to update a value in ϵ which is provided by l-values. The evaluation of expressions to l-values is given in Figure 14, Appendix A. The helper lv_set assigns the l-value's underlying variable (a de Bruin identifier) the new composite value at that location. For instance, lv_set (b `5`[`0`]) `true` ϵ updates the first element of the array to be `true` which is

$$\boxed{\langle \epsilon, e\rangle \Downarrow v}$$

$$\frac{}{\langle \epsilon, b\rangle \Downarrow b}\ \text{E-Bool} \qquad \frac{}{\langle \epsilon, z\langle n\rangle\rangle \Downarrow z\langle n\rangle}\ \text{E-Bit} \qquad \frac{}{\langle \epsilon, z\langle p\rangle\rangle \Downarrow z\langle p\rangle}\ \text{E-Int}$$

$$\frac{\epsilon\ n = v}{\langle \epsilon, \tau\ n\rangle \Downarrow v}\ \text{E-Var} \qquad \frac{\langle \epsilon, e\rangle \Downarrow v}{\langle \epsilon, \ominus\ e\rangle \Downarrow \ominus\ v}\ \text{E-Un}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow v \qquad \text{eval\_slice}\ p_1\ p_2\ v = v'}{\langle \epsilon, e[p_1 : p_2]\rangle \Downarrow v'}\ \text{E-Slice}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow v \qquad \text{eval\_cast}\ \tau\ v = v'}{\langle \epsilon, (\tau)\ e\rangle \Downarrow v'}\ \text{E-Cast}$$

$$\frac{\langle \epsilon, e_1\rangle \Downarrow v_1 \qquad \langle \epsilon, e_2\rangle \Downarrow v_2}{\langle \epsilon, e_1 \oplus e_2\rangle \Downarrow v_1 \oplus v_2}\ \text{E-Bin}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow \overline{v} \qquad n\ \overline{v} = v}{\langle \epsilon, e.n\rangle \Downarrow v}\ \text{E-Mem} \qquad \frac{\overline{\langle \epsilon, e\rangle \Downarrow v}}{\langle \epsilon, \{\overline{e}\}\rangle \Downarrow \{\overline{v}\}}\ \text{E-Lists}$$

$$\frac{\langle \epsilon, e_1\rangle \Downarrow \overline{v} \qquad \langle \epsilon, e_2\rangle \Downarrow z\langle n\rangle \qquad z\ \overline{v} = v}{\langle \epsilon, e_1[e_2]\rangle \Downarrow v}\ \text{E-Index}$$

Figure 7. P4Cub expression evaluation.

Values:

v	::=	b	*boolean*
	|	$z\langle n\rangle$	*unsigned integer*
	|	$z\langle p\rangle$	*signed integer*
	|	$\{\overline{v}\}$	*list*

L-values:

lv	::=	$\tau\ n$	*variable*
	|	$lv[p : p]$	*bit-slicing*
	|	$lv[z]$	*array indexing*
	|	$lv.n$	*struct member*

Evaluated Arguments:

arg_v	::=	in v	*evaluated in-argument*
	|	out lv	*evaluated out-argument*
	|	inout lv	*evaluated inout-argument*

Figure 8. P4Cub value syntax.

sitting at the *fifth* position in ϵ, all other elements of the array remain the same. The E-Assign rule states that after e_2 is fully evaluated to v and e_1 is evaluated to an l-value lv, the location represented by lv in ϵ is updated with a new value, the difference being the component is now represented by v.

As mentioned in Section 3, arguments are specified by *in*, *out*, or *inout*. This matters in evaluating call statements. Arguments specified as *in* are simply input to the procedure, a standard notion of function arguments. Those specified as *out* are evaluated to l-values. Any *out* parameters in function bodies are assigned a value during their evaluation. As in E-FunCall, when copy_out is performed, the value from the function's evaluation environment ϵ' is used to update the call environment ϵ at the location represented by the l-value. For example, suppose some function f has a parameter out b at index 0 and is being applied with an argument out b 1, where 1 is a de Bruijn variable index. The de Bruijn variable is evaluated to the (identical) l-value b 1 by E-LVar. Suppose in the body of f, parameter 0 is assigned to `false`. When the evaluation of f concludes, copy_out looks up that 0 is `false` in ϵ', and assigns 1 to `false` in ϵ. Arguments specified as *inout* serve as both *in* and *out*. E-MtdCall also uses copy_in and copy_out. Because extern methods are externally defined, not in the program syntax, E-MtdCall must make use of ψ and exec_extern to resolve the extern.

Every parser state is a statement block terminated by a well-typed transition *pt* which evaluates to a label. If the label indicates an *intermediate* state, either the start state or a user-defined state, then the appropriate state is looked up and evaluated, conducted by the E-TransI rule. If the label indicates a final state, such as accept (meaning the packet was successfully parsed) or reject (meaning an error in extracting the packet's bits occurred), then the state-machine has concluded evaluating, conducted by the E-TransFinal rule, and control flow goes back to the application of the parser. Both E-TransFinal and E-TransI use the parser transition helper judgment provided in Figure 14, Appendix A. The application of the parser is shown in the E-ApplyP rule and it states that parsers may be applied by other parsers given arguments. As in E-FunCall copy_in and copy_out are used for the arguments to the state-machine.

P4 adopts non-standard scoping conventions. For example, action calls use lexical scope, evident by the E-ActCall rule which looks up both the action's body and a closure environment, that is, $insts\ x = (\epsilon_{cl}, insts', s)$. On the other hand, table invocation and parser transitions use a scheme similar to dynamic scope, evident by the E-Invoke and E-TransI rules that do not use a closure environment. Specifically, E-TransI begins with environment $\epsilon_1 +\!\!+ \epsilon_2$, and the next parser state is then evaluated using ϵ_2 rather than a closure environment, as done in the E-ActCall rule. This evaluation occurs within that of the whole state-machine of a parser with $|\epsilon_2|$ parameters/arguments. Thus when transitioning states in E-TransI only the last $|\epsilon_2|$ values in the environment $\epsilon_1 +\!\!+ \epsilon_2$ should be used when evaluating the next state: ϵ_1 represents variables introduced within the current parser block before the transition takes place. Similarly, in E-Invoke a list append $\epsilon_1 +\!\!+ \epsilon_2$ is used to separate the values in the environment. ϵ_2 is the part of the environment with de Bruijn indices in scope

$$\boxed{\langle \psi, fns, \epsilon, \text{ctx}, s\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}$$

$$\frac{}{\langle \psi, fns, \epsilon, \text{ctx}, \text{skip}\rangle \Downarrow \langle \epsilon, \text{Cont}, \psi\rangle}\ \text{E-Skip} \qquad \frac{}{\langle \psi, fns, \epsilon, \text{ctx}, \text{exit}\rangle \Downarrow \langle \epsilon, \text{Exit}, \psi\rangle}\ \text{E-Exit}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow v}{\langle \psi, fns, \epsilon, \text{ctx}, \text{return } e\rangle \Downarrow \langle \epsilon, \text{Return } v, \psi\rangle}\ \text{E-Rtrn} \qquad \frac{\text{final } l \text{ signal} \qquad \langle \epsilon, pt\rangle \Downarrow_p l}{\langle \psi, fns, \epsilon, \text{Prsr } n\ s\ \overline{s}\ insts, \text{goto } pt\rangle \Downarrow \langle \epsilon, \text{signal}, \psi\rangle}\ \text{E-TransF}$$

$$\frac{\text{get\_state\_block } s\ \overline{s}\ l = s' \qquad \text{intermediate } l \qquad \langle \epsilon_1 +\!\!+ \epsilon_2, pt\rangle \Downarrow_p l \qquad \langle \psi, fns, \epsilon_2, \text{Prsr } |\epsilon_2|\ s\ \overline{s}\ insts, s'\rangle \Downarrow \langle \epsilon_3, \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon_1 +\!\!+ \epsilon_2, \text{Prsr } |\epsilon_2|\ s\ \overline{s}\ insts, \text{goto } p\rangle \Downarrow \langle \epsilon_1 +\!\!+ \epsilon_3, \text{signal}, \psi'\rangle}\ \text{E-TransI}$$

$$\frac{l\langle \epsilon, e_1\rangle \Downarrow_{lv} lv \qquad \langle \epsilon, e_2\rangle \Downarrow v}{\langle \psi, fns, \epsilon, \text{ctx}, e_1 := e_2\rangle \Downarrow \langle \text{lv\_set } lv\ v\ \epsilon, \text{Cont}, \psi\rangle}\ \text{E-Assign}$$

$$\frac{fns\ x = fns', s \qquad l\langle \epsilon, e\rangle \Downarrow_{lv} lv \qquad \overline{\langle \epsilon, arg\rangle \Downarrow_{arg} arg_v} \qquad \langle \psi, fns', \text{copy\_in } \overline{arg_v}\ \epsilon, \text{Function}, s\rangle \Downarrow \langle \epsilon', \text{Return } v, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, e\ x\langle\overline{\tau}\rangle(\overline{arg})\rangle \Downarrow \langle \text{lv\_set } lv\ v\ (\text{copy\_out } \overline{arg_v}\ \epsilon'\ \epsilon), \text{Cont}, \psi'\rangle}\ \text{E-FunCall}$$

$$\frac{insts\ x = (\epsilon_{cl}, insts', s) \qquad \overline{\langle \epsilon, e\rangle \Downarrow v} \qquad \overline{\langle \epsilon, arg\rangle \Downarrow_{arg} arg_v} \qquad \langle \psi, fns, \overline{v} +\!\!+ \text{copy\_in } \overline{arg_v}\ \epsilon_{cl}, \text{Ctrl}\ \ insts', s\rangle \Downarrow \langle \epsilon', \text{Return}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{Ctrl}\ \ insts, x\ (\overline{e}, \overline{arg})\rangle \Downarrow \langle \text{copy\_out } \overline{arg_v}\ \epsilon'\ \epsilon, \text{Cont}, \psi'\rangle}\ \text{E-ActCall}$$

$$\frac{l\langle \epsilon, e\rangle \Downarrow_{lv} lv \qquad \overline{\langle \epsilon, arg\rangle \Downarrow_{arg} arg_v} \qquad \text{exec\_extern } \psi\ x\ x_m\ \overline{\tau}\ \overline{arg_v} = (v, \overline{arg_v}', \psi')}{\langle \psi, fns, \epsilon, \text{ctx}, e\ x\ x_m\langle\overline{\tau}\rangle(\overline{arg})\rangle \Downarrow \langle \text{lv\_set } lv\ v\ (\text{copy\_out } \overline{arg_v}'\ \epsilon), \text{Cont}, \psi'\rangle}\ \text{E-MtdCall}$$

$$\frac{insts\ x_t = (|\epsilon_2|, \overline{e}_k, \overline{(x, \overline{arg})}) \qquad \text{match\_actions } \psi\ \overline{e}_k\ \overline{(x, \overline{arg})} = x_a, \overline{e}, \overline{arg} \qquad \langle \psi, fns, \epsilon_2, \text{Ctrl}\ \ insts, x_a\ (\overline{e}, \overline{arg})\rangle \Downarrow \langle \epsilon', \text{Cont}, \psi'\rangle}{\langle \psi, fns, \epsilon_1 +\!\!+ \epsilon_2, \text{Ctrl}\ \ insts, \text{invoke } x_t\rangle \Downarrow \langle \epsilon_1 +\!\!+ \epsilon', \text{Cont}, \psi'\rangle}\ \text{E-Invoke}$$

$$\frac{insts\ x = fns', insts', s \qquad \overline{\langle \epsilon, arg\rangle \Downarrow_{arg} arg_v} \qquad \langle \psi, fns', \text{copy\_in } \overline{arg_v}\ \epsilon, \text{Ctrl}\ \ insts', s\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{Ctrl}\ \ insts, \text{apply } x(\overline{arg})\rangle \Downarrow \langle \text{copy\_out } \overline{arg_v}\ \epsilon'\ \epsilon, \text{Cont}, \psi'\rangle}\ \text{E-ApplyC}$$

$$\frac{insts\ x = fns', insts', s', \overline{s}' \qquad \langle \psi, fns', \text{copy\_in } \overline{arg_v}\ \epsilon, \text{Prsr } |\overline{arg}|\ s'\ \overline{s}'\ insts', s'\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{Prsr } n\ s\ \overline{s}\ insts, \text{apply } x(\overline{arg})\rangle \Downarrow \langle \text{copy\_out } \overline{arg_v}\ \epsilon'\ \epsilon, \text{Cont}, \psi'\rangle}\ \text{E-ApplyP}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow v \qquad \langle \psi, fns, v :: \epsilon, \text{ctx}, s\rangle \Downarrow \langle v' :: \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, \text{let } e \text{ in } s\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}\ \text{E-LetIn} \qquad \frac{\text{interrupt signal} \qquad \langle \psi, fns, \epsilon, \text{ctx}, s_1\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, s_1; s_2\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}\ \text{E-SeqI}$$

$$\frac{\langle \psi, fns, \epsilon, \text{ctx}, s_1\rangle \Downarrow \langle \epsilon', \text{Cont}, \psi'\rangle \qquad \langle \psi', fns, \epsilon', \text{ctx}, s_2\rangle \Downarrow \langle \epsilon'', \text{signal}, \psi''\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, s_1; s_2\rangle \Downarrow \langle \epsilon'', \text{signal}, \psi''\rangle}\ \text{E-SeqC} \qquad \frac{\langle \epsilon, e\rangle \Downarrow \texttt{true} \qquad \langle \psi, fns, \epsilon, \text{ctx}, s_1\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, \text{if } e \text{ then } s_1 \text{ else } s_2\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}\ \text{E-CondT}$$

$$\frac{\langle \epsilon, e\rangle \Downarrow \texttt{false} \qquad \langle \psi, fns, \epsilon, \text{ctx}, s_2\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}{\langle \psi, fns, \epsilon, \text{ctx}, \text{if } e \text{ then } s_1 \text{ else } s_2\rangle \Downarrow \langle \epsilon', \text{signal}, \psi'\rangle}\ \text{E-CondF}$$

Figure 9. P4Cub statement evaluation.

at the table's definition. ϵ_1 represents variables introduced after the table declaration. To ensure any de Bruijn indices in the data plane arguments look up the correct values in the environment $\epsilon_1 +\!\!+ \epsilon_2$, a suffix ϵ_2 of the environment is used, whose length is equal to the number of variables in scope at the syntactic place of the table. Since the table's declaration, we have that $|\epsilon_1|$ variables have been declared in the control.

5 Implementation

P4Cub's Coq implementation itself runs to roughly 7,400 lines of code and uses Petr4 [Doenges et al. 2021] as a front-end for the lexer, parser, and type checker. P4Cub is divided into modules for syntax, semantics, and program transformations. P4Cub syntax and semantics are essentially complete but do have a few relatively minor limitations. These limitations do not preclude using P4Cub for real-world programs and we expect addressing them will be straightforward. In the future, we hope to prove many properties for statements such as type soundness and semantic preservation for different stages of the compiler.

6 Case Studies

To evaluate our design for P4Cub, we present a series of case studies using the language to perform a variety of tasks. In Section 6.1 and Section 6.2 we study how de Bruijn indices improve both proof and code quality, by exploring type system metatheory and a compiler pass respectively. Finally, in Section 6.3, we describe a prototype verifier, and observe how the streamlined P4Cub syntax simplifes the effort.

6.1 Metatheory

We have proven preservation and progress of the big-step evaluation of expressions.

Theorem 7. *Expression evaluation preserves typing.*

$$\forall\ \epsilon\ e\ v\ \Gamma\ \tau,\\ \langle \epsilon, e\rangle \Downarrow v \to \overline{\vdash_v\ \epsilon : \Gamma} \to \Gamma\ \vdash\ e : \tau \to \vdash_v\ v : \tau$$

Theorem 8. *A well-typed expression will evaluate.*

$$\forall\ \Gamma\ e\ \tau\ \epsilon, \overline{\vdash_v\ \epsilon : \Gamma} \to \Gamma\ \vdash\ e : \tau \to \exists\ v, \langle \epsilon, e\rangle \Downarrow v$$

Furthermore we have shown preservation and progress hold for l-expression evaluation. Here we can see the dividends of our choice to use de Bruijn indices—each of these theorems has a premise $\overline{\vdash_v\ \epsilon : \Gamma}$, which indicates that all of the values in the store ϵ have type Γ at the same de Bruijn index. This ensures that when evaluating a variable, its corresponding value in the store preserves its type. We have found this to be a much easier way to relate the typing Γ and evaluation ϵ as opposed to having mappings from strings to types or values. $\overline{\vdash_v\ \epsilon : \Gamma}$ succinctly indicates both that Γ and ϵ have the same domain of (de Bruijn) variable names and that their elements type correspondingly.

Expression evaluation is also deterministic:

Theorem 9. *Determinism.*

$$\forall\ \epsilon\ e\ v_1\ v_2, \langle \epsilon, e\rangle \Downarrow v_1 \to \langle \epsilon, e\rangle \Downarrow v_2 \to v_1 = v_2$$

In Coq it looks like:

```
Theorem expr_deterministic : forall ϵ e v1 v2,
⟨ ϵ, e ⟩ ⇓ v1 ->
⟨ ϵ, e ⟩ ⇓ v2 -> v1 = v2.
Proof.
intros eps e v1 v2 Hv1; generalize dependent v2;
induction Hv1 using custom_expr_big_step_ind;
intros V2 HV2; inv HV2; f_equal; auto 4.
pose proof Forall2_forall_impl_Forall2
_ _ _ _ _ _ _ H0 _ H4 as h.
rewrite Forall2_eq in h; assumption.
Qed.
```

In the future, we plan to prove analogous properties for statement evaluation, completing type soundness proofs for the full big-step semantics. We have verified a few auxiliary properties for statement evaluation, such as the following.

Theorem 10. *The de Bruijn store's length is preserved by statement evaluation.*

$$\forall\ fns\ \psi\ \psi'\ \epsilon\ \epsilon'\ \mathsf{ctx}\ s\ \mathsf{signal},\\ \langle \psi, fns, \epsilon, \mathsf{ctx}, s\rangle \Downarrow \langle \epsilon', \mathsf{signal}, \psi'\rangle \to |\epsilon| = |\epsilon'|$$

This property ensures that de Bruijn indices have the same meaning before and after a statement is evaluated. For our full theorem of statement preservation we will hope to show that input and output stores type as the same Γ. This proof has been automated in Coq.

```
Lemma sbs_length : forall Ψ ϵ ϵ' c s sig ψ,
⟨ Ψ, ϵ, c, s ⟩ ⇓ ⟨ ϵ', sig, ψ ⟩
-> length ϵ = length ϵ'.
Proof using.
  intros ? ? ? ? ? ? ? h;
    induction h; autorewrite with core in *; auto; lia.
Qed.
```

Proving progress of statement big-step evaluation will require reasoning about program termination. Parser state machines in particular may prove difficult. We hope to build on work such as Leapfrog [Doenges et al. 2022], which is implementing powerful tools to reason about packet-parsing state machines, and perhaps adopt their methods to formally verify properties of P4Cub parsers. Nevertheless, even this initial case study, mechanized in Coq, demonstrates the utility of the P4Cub IR for formal reasoning.

6.2 Compiler Passes

We are currently building a compiler from P4Cub to Clight. We hope to be able to verify semantics-preservation for each translation between IRs. P4Cub and Clight both require function calls to take place at the statement level. However, C does not have numeric data-types such as P4's bit$\langle n\rangle$ and int$\langle p\rangle$ for arbitrary bit-length n or p respectively. C only supports specific sizes for unsigned and signed integers. To translate to C we must use a bit-vector library that generates P4 integer literals as function calls in Clight. This means

literals such as $z\langle n\rangle$ must be moved to the statement level in-order to be compiled to Clight.

We have implemented a pass from P4Cub to P4Cub to lift such terms to the top-level of expressions. This pass has been verified to produce actually "lifted" terms, and has been shown to preserve both expression typing and evaluation. The implementation and correctness specifications for this pass influenced our decision to adopt a de Bruijn convention for term variables. We found the specification to be much more elegant and the proofs more tractable than those of a standard naming convention.

The lifting pass for expressions, represented by the judgment $(e \Uparrow \overline{e}, e')$, works by generating both a new "lifted" term, e', as well as list of terms that will become variable declarations, $\overline{e}$. This pass performs any necessary de Bruijn shifts on resultant and intermediate terms. If a term needs to be entirely lifted to the statement level it is replaced with a variable of index 0, and the lifted term is pushed to the stack of lifted terms to become variable declarations. This unwinding of lifted term variables occurs at the statement-level, where all of the variable declarations envelope the block for which these variables will be in scope.

The specification of the correctness theorem uses a relation between the right-hand-side terms-to-be and their values. The statement uses the relation eval_decl_list $\epsilon\ \overline{e}\ \overline{v}$, which says that in context ϵ, $\overline{e}$ evaluates to $\overline{v}$.

Theorem 11.

$$\forall\ \epsilon\ e\ e'\ \overline{e}\ v, \langle \epsilon, e\rangle \Downarrow v \rightarrow e \Uparrow \overline{e}, e' \rightarrow \exists\ \overline{v}, \text{eval\_decl\_list}\ \epsilon\ \overline{e}\ \overline{v} \wedge \langle \overline{v} +\!\!+ \epsilon, e'\rangle \Downarrow v$$

In English, this theorem shows that lifted terms evaluate to the same value as the original. However, when a term is lifted it produces a sequence of other terms. This sequence of terms will become a series of embedded variable declarations let e_1 in let e_2 in ...e'..., where e' is the lifted version of the original term. Therefore the environment to evaluate e' will also depend upon the series of variable declarations. This *unwinding* of the list $\overline{e}$ in the specification is expressed as eval_decl_list $\epsilon\ \overline{e}\ \overline{v}$, and it gives us the appropriate environment to evaluate the lifted term. We have further proven that evaluation is preserved after lifted terms are unwound in the corresponding statement.

We are working to show that the lifting pass correctly preserves such properties for other levels of P4Cub syntax. Statements have proved to be particularly challenging but we hope to soon fully prove the lifting pass preserves statement evaluation. Subgoals for cases such as variable declarations are promising but there is still work to be done.

6.3 A Program Verifier

We have prototyped a program verifier for P4Cub programs *à la* p4v [Liu et al. 2018], Aquila [Tian et al. 2021], and Vera [Stoenescu et al. 2018]. The core of this verifier is a compiler from P4Cub to Dijkstra's Guarded Command Logic (GCL) [Dijkstra 1975]. Targeting a well-understood calculus allows us to use standard verification algorithms instead of having to reimplement them from scratch for P4Cub.

The design of the compiler is shown in Figure 10. It is a two-pass compiler from P4Cub to GCL via another IR called `Inline`. The `Inline` IR is like P4Cub in every way except that all invocations of abstractions (extern methods, parser transitions, tables, actions, and applications) are replaced with their definitions.

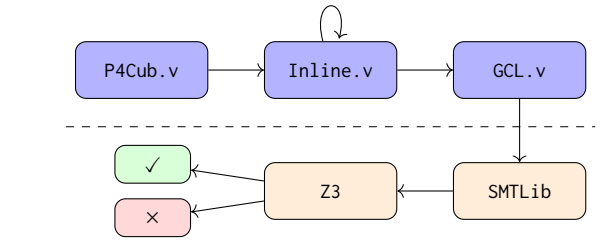
Implementing this pass required navigating with Coq's notoriously conservative termination checker. Replacing function names with substituted function bodies, for instance, could certainly run forever if P4Cub programs contained recursive calls. Rather than prove this, we add a `gas` parameter to the inlining function to temporarily bypass the termination checker. Using a separate AST lets us quarantine this termination bypass in our code.

The only place where recursion may truly exist is in the parser—a common design pattern for parsing header stacks is to use a state with a self-loop. Fortunately, the P4 language specification [P4 Language Consortium 2022] requires parser loops to be finitely unrollable. So we can get away with providing an additional `unroll` parameter that specifies how many times to unroll the parser. There's a subtle difference between the `unroll` and `gas` parameters—running `gas` to `0` triggers a compilation failure, prompting the user to try again with more, while running `unroll` to `0` causes the parser-inliner to stop unrolling.

One advantage of keeping the core parser logic in P4Cub is that verifiers can choose different representation strategies for parsers. In certain domains (e.g. verification), we've found it advantageous to use Aquila's encoding optimization [Tian et al. 2021], however in others (e.g., certain synthesis tasks), the preponderance of new variables it introduces can be costly. Leaving the parser in the IR lets us choose our encoding based on the task at hand. In the verifier we use Aquila's encoding trick, which avoids the potential blowup of naively inlining each state [Tian et al. 2021]. Each state s (including `accept` and `reject`) is given a 1-bit ghost variable `_state$s$next`, which is 1 when s is the next state to be executed. Then transitions amount to setting the appropriate bits and the unrolled states can be printed sequentially.

After inlining, we perform a few elimination passes, which is where the streamlined nature of P4Cub really shines. For example, rather than writing separate elimination logic for lists, structs, headers, and arrays, we can handle them all with a single case. Once the program has been reduced to solely use bitvector expressions, we can compile the statements to GCL directly as done in p4v [Liu et al. 2018].

Retaining tables in P4Cub leaves verifiers freedom in modeling tables. Tables can be compiled away using ghost variables [Liu et al. 2018]. However, different ghost variable models are appropriate for different verification tasks [Campbell




```
parser MyParser(packet_in packet,
    out headers hdr, inout metadata meta,
    inout standard_metadata_t standard_metadata) {
  state start {
    packet.extract(hdr.ethernet);
    transition accept;            }  }
control MyIngress(inout headers hdr, inout metadata meta,
    inout standard_metadata_t standard_metadata) {
  action drop() { mark_to_drop(standard_metadata); }
  action fwd(bit<9> port) {
    standard_metadata.egress_spec = port;}
  table sw {
    key = { hdr.ethernet.dstAddr: exact; }
    actions = { fwd; drop; } }
  apply { sw.apply(); }}
```

```
_state$accept$next := (_ bv0 1);
_state$reject$next := (_ bv0 1);
_state$start$next := (_ bv1 1);
{ assume (not (= _state$start$next (_ bv1 1)))
} [] { assume (= _state$start$next (_ bv1 1));
       _state$start$next := (_ bv0 1);
       hdr.ethernet.is_valid := (_ bv1 1);
       _state$accept$next := (_ bv1 1)
};
{ assume (not (= _state$accept$next (_ bv1 1))) } [] {
  assume (= _state$accept$next (_ bv1 1));
  assert (= hdr.ethernet.is_valid (_ bv1 1));
  assume (= _symb$sw$match_0 hdr.ethernet.dstAddr);
  { assume (= _symb$sw$action (_ bv0 1));
    _return$sw.action_run := (_ bv0 1);
    standard_metadata.egress_spec := (_ bv511 9)
  } [] {
    assume (= _symb$sw$action (_ bv1 1));
    _return$sw.action_run := (_ bv1 1);
    standard_metadata.egress_spec :=
       _symb$sw$fwd$arg$port
  }
}
```

Figure 10. Compilation from P4 surface syntax (bottom left) to GCL (right). Top left shows the compiler design; modules above the dotted line are extracted to OCaml, modules below the line are written in OCaml.

et al. 2021; Eichholz et al. 2022; Stoenescu et al. 2018]. The choice of table model affects compilation and verification condition generation. To allow for these various backend approaches we parameterize the compiler module with a function `Variable` called `instr`, which maps table data (name, keys, and actions) to an implementation. This allows users of the verification tool to plug in the table model most fitting for their analysis.

For example, Figure 10 shows a particular implementation choice for a single table `sw`. We use ghost variables to symbolically represent the runtime contents of the table. The variable `_symb$sw$action` symbolically represents the controller's action choice, `_symb$sw$fwd$arg$port` represents the `port` action data variable for the `fwd` action in table `sw`, and `_symb$sw$match_0` represents the 0th match key in table `sw`. We then `assume` that these are equal to the relevant values, leveraging nondeterminism to capture the full range of possible table states.

Finally, we extract all of the modules and tie them together in OCaml to build our program verifier. In OCaml, we convert our GCL program into an SMT-LIB term using a standard verification condition generation algorithm [Dijkstra 1975; Flanagan and Saxe 2001], and pass that term to Z3, which determines whether it is valid (✓) or invalid (×).

Figure 10 shows an example verification problem we might pose to a verifier. We'll highlight a few aspects of the translation. Here, we're checking that the undefined value triggered by accessing invalid headers never arises.[2] In the source P4 program, the only header-data read occurs in the `key` clause of the table `sw`, where we read the `dstAddr` field of the `hdr.ethernet` header. In the compiled GCL code on the right of the figure, this read is translated into two statements (highlighted in blue): the first asserts that the `hdr.ethernet` header is valid (which crashes rather than producing an undefined value), and the second assumes that the header field `hdr.ethernet.dstAddr` is equivalent to `_symb$sw$match_0`, which symbolically represents the table's match values. Note that GCL does not have structured data or headers: each dot in `hdr.ethernet.is_valid` is not an operator, it is just part of the identifier.

To manually prove that this assertion is never violated, we can examine the translation of the parser. Observe that `hdr.ethernet` is `extracted` in the `start` state, which is always executed. In corresponding the GCL code, we set the validity bit for the Ethernet header to `1`, which will prove the assertion. To prove this automatically, we compute a standard quadratic-size verification condition [Flanagan and Saxe 2001] in SMT-LIB, and check its validity using Z3.

7 Related Work

We briefly survey the most relevant related work to P4Cub, focusing on IRs, certified frameworks, and P4 verification.

[2]This so-called *header validity problem* is akin to the pointer nullability problem in Java or C, and has been heavily studied [Banerjee et al. 2019; Eichholz et al. 2019, 2022; Liu et al. 2018; Stoenescu et al. 2018].

Intermediate Representations. LLVM [Lattner and Adve 2004] is perhaps the most well-known modern compiler IR—its SSA abstraction allows for efficient compilation of many languages. One of the more notable success stories is the Clang compiler [Lattner 2008] for C/C++ and Objective-C. The MLIR project [Lattner et al. 2020] evolved from LLVM as a general purpose IR targeting domain-specific languages, including in machine learning, with built-in abstractions for domain-specific customization. ILA [Huang et al. 2018], like P4Cub, is meant to be a low-level IR for special-purpose hardware targets but, unlike P4Cub, is meant for heterogeneous hardware accelerators rather than network programs.

Certified Frameworks. P4Cub also draws inspiration from Coq frameworks like CompCert [Leroy et al. 2016], a C compiler with a fully mechanized semantics preservation proof for a subset of the C language. In the future, we plan to prove similar correctness theorems for P4Cub's various backends. The Vellvm project [Zhao et al. 2012] provides a formal semantics for a subset of LLVM, to facilitate the development of certified LLVM compilers. Finally, the Verified Software Toolchain [Appel 2011] is an ongoing project developing static analyzers, program verifiers, and compilers for the C programming language, including program logics like Verifiable C [Appel et al. 2016].

P4 Verification. Petr4 [Doenges et al. 2021] and P4K [Kheradmand and Rosu 2018] have both defined formal semantics for the P4 language, while Petr4 realized P4's type system and proved it sound. Other type systems and formal models for P4 have been explored [Eichholz et al. 2019, 2022], though none suffice as compiler IRs, being themselves highly idealized versions of the language. P4v [Liu et al. 2018], Aquila [Tian et al. 2021], Vera [Stoenescu et al. 2018], and Assert-P4 [Freire et al. 2018] are P4 program verifiers that translate P4 to GCL to compute verification conditions. The closest lines of work to P4Cub are Verifiable P4 [Wang et al. 2022] and HOL-P4 [Alshnakat et al. 2022]. Verifiable P4 is a program logic for proving properties of P4 programs in Coq. It operates on a slightly higher-level IR, P4light, which resembles P4 surface syntax more closely than P4Cub. P4light is a good fit for a program logic meant to verify programs as they are written, but this fidelity to surface P4 makes it more awkward than P4Cub for compilation and automated verification. HOL-P4 is a contemporaneous mechanization of P4 using the HOL4 theorem prover. Like P4Cub, it uses Petr4 as a front-end, but adopts a different approach to modeling the semantics—e.g., it uses a stack rather than a heap.

8 Conclusion and Future Work

P4Cub is a new mechanized IR for P4 that provides a clean foundation for building certified tools. It is available as an open source project on GitHub under the Apache2 license, and is intended to be a resource for the entire community. In the future, we plan to continue building on P4Cub, including developing an verified compiler that uses CompCert as a backend. We plan to explore formalizing various standard protocols in P4, using P4Cub to obtain a fully-verified reference implementation. Finally, we hope to work with the P4 Language Design Working Group to get P4Cub's semantics adopted as a companion to the official language specification.

Acknowledgments

The authors wish to thank the CPP '23 reviewers for helpful suggestions, and Pico and Leo for their support. This work was funded in part by DARPA under contract HR0011-20-C-0107 and by the NSF under grant FMiTF-1918396.

A Supporting Judgments

Control declarations:

cd	::=	var e	*local*
	|	action $x(\overline{\tau})(\overline{prm})\ \{s\}$	*action*
	|	table x {key = $\overline{e}$; actions = $\overline{(x, \overline{arg})}$}	*table*

Figure 11. P4Cub declarations within controls.

Top-level declarations:

td	::=	instance x of $x(\overline{\tau})$	*instantiate*
	|	extrn $x\langle n\rangle\{\overline{\tau\ x\langle n\rangle(\overline{prm})}\}$	*extern*
	|	ctrl $x(\overline{prm})\{\overline{cd}\}$ apply s	*controls*
	|	prsr $x(\overline{prm})$ start = s $\{\overline{s}\}$	*parsers*
	|	$\tau\ x\langle n\rangle(\overline{prm})\{s\}$	*functions*

Figure 12. P4Cub top-level program declarations.

B Declaration Syntax

As shown in Figure 11 and Figure 12, P4Cub distinguishes between declarations that may occur within control blocks, denoted by cd, and those that occur at the top-level of a program, denoted by td. P4Cub control declarations include (de Bruijn) local variable declarations, and actions and tables which represent the eponymous constructs of match-action tables. Actions interface with the control-plane of switches, and as such have parameters for the control-plane and those for the data plane. Control-plane parameters are given as $\overline{\tau}$, and data plane parameters are given as $\overline{prm}$. P4Cub table declarations include a *key* provided by $\overline{e}$, which is used by the control-plane as input to determine which action to call. They are also used to determine control-plane arguments. The actions field $\overline{(x, \overline{arg})}$ names actions to call. Each action name is paired with data plane arguments provided by the programmer.

Parser transition typing:

$$\boxed{n, \Gamma \vdash_p pt}$$

$$\frac{\text{valid\_state } n\ l}{n, \Gamma \vdash_p \text{direct } l}\ \text{T-DirectTrans}$$

$$\frac{\Gamma \vdash e : \tau \qquad \text{valid\_state } n\ l \qquad \overline{pat} : \tau \qquad \text{valid\_state } n\ \overline{l}}{n, \Gamma \vdash_p \text{select } e\ l\ \{\overline{pat \Rightarrow l}\}}\ \text{T-SelectTrans}$$

Pattern typing:

$$\boxed{pat : \tau}$$

$$\frac{}{\_ : \tau}\ \text{T-Wild} \qquad \frac{}{z\langle n \rangle : \text{bit}\langle n \rangle}\ \text{T-BitPat}$$

$$\frac{}{z\langle p \rangle : \text{int}\langle p \rangle}\ \text{T-IntPat} \qquad \frac{pat_1 : \text{bit}\langle n \rangle \quad pat_2 : \text{bit}\langle n \rangle}{pat_1 \ \&\&\& \ pat_2 : \text{bit}\langle n \rangle}\ \text{T-Mask}$$

$$\frac{pat_1 : \text{bit}\langle n \rangle \quad pat_2 : \text{bit}\langle n \rangle}{pat_1 \ .. \ pat_2 : \text{bit}\langle n \rangle}\ \text{T-Range}$$

$$\frac{\overline{pat : \tau}}{\overline{pat} : \overline{\tau}}\ \text{T-ListPat}$$

Argument typing:

$$\boxed{\Gamma \vdash_{arg} arg : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{arg} \text{in } e : \tau}\ \text{T-In} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{arg} \text{out } e : \tau}\ \text{T-Out}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{arg} \text{out } e : \tau}\ \text{T-Inout}$$

Figure 13. Helper judgments for P4Cub statement typing.

Parser transition expression evaluation:

$$\boxed{\langle \epsilon, pt \rangle \Downarrow_p l}$$

$$\frac{}{\langle \epsilon, \text{direct } l \rangle \Downarrow_p l}\ \text{E-DirTrns}$$

$$\frac{\langle \epsilon, e \rangle \Downarrow v \qquad v \text{ matches } pat \Rightarrow l' \in \overline{pat \Rightarrow l}}{\langle \epsilon, \text{select } e\ l\ \{\overline{pat \Rightarrow l}\} \rangle \Downarrow_p l'}\ \text{E-TrnsMtch}$$

$$\frac{\langle \epsilon, e \rangle \Downarrow v \qquad v \text{ has no matches in } \overline{pat \Rightarrow l}}{\langle \epsilon, \text{select } e\ l\ \{\overline{pat \Rightarrow l}\} \rangle \Downarrow_p l}\ \text{E-TrnsDflt}$$

Argument evaluation:

$$\boxed{\langle \epsilon, arg \rangle \Downarrow_{arg} arg_v}$$

$$\frac{\langle \epsilon, e \rangle \Downarrow v}{\langle \epsilon, \text{in } e \rangle \Downarrow_{arg} \text{in } v}\ \text{E-In} \qquad \frac{l\langle \epsilon, e \rangle \Downarrow_{lv} lv}{\langle \epsilon, \text{out } e \rangle \Downarrow_{arg} \text{out } lv}\ \text{E-Out}$$

$$\frac{l\langle \epsilon, e \rangle \Downarrow_{lv} lv}{\langle \epsilon, \text{inout } e \rangle \Downarrow_{arg} \text{inout } lv}\ \text{E-Inout}$$

L-value evaluation:

$$\boxed{l\langle \epsilon, e \rangle \Downarrow_{lv} lv}$$

$$\frac{}{l\langle \epsilon, \tau\ n \rangle \Downarrow_{lv} \tau\ n}\ \text{E-LVar}$$

$$\frac{l\langle \epsilon, e \rangle \Downarrow_{lv} lv}{l\langle \epsilon, e[p_1 : p_2] \rangle \Downarrow_{lv} lv[p_1 : p_2]}\ \text{E-LSlice}$$

$$\frac{l\langle \epsilon, e \rangle \Downarrow_{lv} lv}{l\langle \epsilon, e.n \rangle \Downarrow_{lv} lv.n}\ \text{E-LMem}$$

$$\frac{l\langle \epsilon, e_1 \rangle \Downarrow_{lv} lv \qquad \langle \epsilon, e_2 \rangle \Downarrow z\langle n \rangle}{l\langle \epsilon, e_1[e_2] \rangle \Downarrow_{lv} lv[z]}\ \text{E-LIdx}$$

Figure 14. Helper judgments for statement evaluation.

The top-level instantiates are controls, parsers, externs, and the "main" pipeline itself. Notice that unlike P4, P4Cub disallows nested `parser` and `control` instantiations and instead requires them to be instantiated at the top level—nested instantiations do not increase the expressiveness of the language, and they can always be inlined. For example, P4Cub only allows the second `control c` definition in Example 2.

These instances are those applied in apply statements, as well as the externs used in method calls. Like P4, P4Cub extern declarations are merely a signature of the functionality provided by the underlying target architecture. Extern declarations provide the methods' signatures that are available for the programmer to call. Declarations of controls are composed of a list of *cd* (actions and tables), with a final *s* representing the *apply block* of the control. This apply block is a *main* of the control: when a control is applied this is the statement that is executed. Parsers specify a start state, as well as a list of user-defined states. The list of states are labeled by a natural number, and each statement is expected to conclude with a transition statement. A P4Cub program is a list of declarations *td*.

Figure 13 provides the auxiliary judgments used in typing of statements, shown in Figure 6. The parser transition typing judgment determines if the transition *pt* in a parser with *n* number of states is valid under the environment Γ.

$$\boxed{e \Uparrow \overline{e}, e'}$$

$$\frac{}{\tau\ n \Uparrow [], \tau\ n}\ \text{L-Variable} \qquad \frac{}{\text{err}\ x \Uparrow [], \text{err}\ x}\ \text{L-Error}$$

$$\frac{}{b \Uparrow [], b}\ \text{L-Bool}$$

$$\frac{e \Uparrow \overline{e}, e'}{e.n \Uparrow \overline{e}, e'.n}\ \text{L-Mem} \qquad \frac{}{z\langle n\rangle \Uparrow [z\langle n\rangle], \text{bit}\langle n\rangle\ 0}\ \text{L-Bit}$$

$$\frac{}{z\langle p\rangle \Uparrow [z\langle p\rangle], \text{int}\langle p\rangle\ 0}\ \text{L-Int}$$

$$\frac{e \Uparrow \overline{e}, e'}{e[p_1 : p_2] \Uparrow e[p_1 : p_2] :: \overline{e}, (\text{bit}\langle p_1 - p_2 + 1\rangle)\ 0}\ \text{L-Slice}$$

$$\frac{e \Uparrow \overline{e}, e'}{(\tau)\ e \Uparrow (\tau)\ e :: \overline{e}, \tau\ 0}\ \text{L-Cst}$$

$$\frac{e \Uparrow \overline{e}, e' \qquad [] \vdash e : \tau}{\ominus e \Uparrow \ominus e :: \overline{e}, \tau\ 0}\ \text{L-Un}$$

$$\frac{\begin{array}{c} e \Uparrow \overline{e_1}, e_1' \qquad e \Uparrow \overline{e_2}, e_2' \\ [] \vdash e_1 \oplus e_2 : \tau \qquad \uparrow^0_{|\overline{e_1}|} \overline{e_2} +\!\!+ \overline{e_1} = \overline{e} \\ \uparrow^0_{|\overline{e_2}|} e_1' \oplus \uparrow^{|\overline{e_2}|}_{|\overline{e_1}|} e_2' = e' \end{array}}{e_1 \oplus e_2 \Uparrow e' :: \overline{e}, \tau\ 0}\ \text{L-Bin}$$

$$\frac{e \Uparrow \overline{e_1}, e_1' \qquad e \Uparrow \overline{e_2}, e_2'}{e_1[e_2] \Uparrow \uparrow^0_{|\overline{e_1}|} \overline{e_2} +\!\!+ \overline{e_1}, \uparrow^0_{|\overline{e_2}|} e_1'[\uparrow^{|\overline{e_2}|}_{|\overline{e_1}|} e_2']}\ \text{L-Idx}$$

$$\frac{\begin{array}{c} \overline{e \Uparrow \overline{e'}, e''} \\ \text{shift\_pairs}\ \overline{\overline{e'}\ \overline{e''}} = (\overline{e'''}, \overline{e''''}) \qquad [] \vdash \{\overline{e}\} : \tau \end{array}}{\{\overline{e}\} \Uparrow \{\overline{e'''}\} :: \overline{e''''}, \tau\ 0}\ \text{L-List}$$

Figure 15. The lifting pass of expressions.

The pattern typing judgment states that the pattern *pat* has the type τ. The helper function valid_state $n\ l$ determines if the state l is valid or not given the total states n of a parser. The start, accept, and reject states are valid. Additionally, in P4Cub, user-defined parser-states are labeled with natural numbers, thus, a valid reference to such a state must be less than the parser's total number of states n. Finally, the argument typing judgment determines the type of an argument.

Figure 14 provides the auxiliary judgments used in evaluating statements, shown in Figure 9. The parser transition evaluation simply evaluates the parser transition expression. Note that in P4Cub, as in P4, the actual work of extracting

$$\boxed{\text{eval\_decl\_list}\ \epsilon\ \overline{e}\ \overline{v}}$$

$$\frac{}{\text{eval\_decl\_list}\ \epsilon\ []\ []}\ \text{E-DclNil}$$

$$\frac{\langle \overline{v} \dashv\!\!\dashv \epsilon, e\rangle \Downarrow v \qquad \text{eval\_decl\_list}\ \epsilon\ \overline{e}\ \overline{v}}{\text{eval\_decl\_list}\ \epsilon\ (e :: \overline{e})\ (v :: \overline{v})}\ \text{E-DclCns}$$

Figure 16. Evaluation of lifted list.

metadata from headers is done by externs, and is opaque in our definitions here. The argument evaluation determines whether to evaluate an argument to a value or a l-value. This is needed because inter-procedural calls in P4 such as those to functions and actions have a copy-in and copy-out semantics. Lastly, some expressions are partially evaluated to l-values instead of values. The l-value evaluation provides such rules.

C Lifting Compiler Pass

Figure 15 describes the compiler pass for lifting out complex expressions. Figure 16 describes how to evaluate the declaration lists.

References

Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. 2022. SwitchV: Automated SDN Switch Validation with P4 Models. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 365–379. https://doi.org/10.1145/3544216.3544220

Anoud Alshnakat, Didrik Lundberg, Roberto Guanciale, Mads Dam, and Karl Palmskog. 2022. HOL4P4: Semantics for a Verified Data Plane. In *Proceedings of the International Workshop on P4 in Europe*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3565475.3569081

Andrew W Appel. 2011. Verified software toolchain. In *European Symposium on Programming*. Springer, 1–17. https://doi.org/10.1007/978-3-642-28891-3_2

Andrew W Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. 2016. *Verifiable C*. https://doi.org/10.1007/s10817-018-9457-5

Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 740–750. https://doi.org/10.1145/3338906.3338919

Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 133–153.

Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations

for p4 data planes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021). https://doi.org/10.1145/3434322

Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: Certified Equivalence for Protocol Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3519939.3523715

Matthias Eichholz, Eric Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. 2019. How to avoid making a billion-dollar mistake: Type-safe data plane programming with SafeP4. *ECOOP* (2019). https://doi.org/10.48550/ARXIV.1906.07223

Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-typed data plane programming. *POPL* (2022). https://doi.org/10.1145/3498701

Cormac Flanagan and James B Saxe. 2001. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 193–205. https://doi.org/10.1145/373243.360220

Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*. 1–7. https://doi.org/10.1145/3185467.3185499

Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (dec 2018), 24 pages. https://doi.org/10.1145/3282444

Ali Kheradmand and Grigore Rosu. 2018. P4K: A formal semantics of P4 and applications. *arXiv preprint arXiv:1804.01468* (2018). https://doi.org/10.48550/arXiv.1804.01468

Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://dl.acm.org/doi/abs/10.5555/977395.977673

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020). https://doi.org/10.48550/ARXIV.2002.11054

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*. 490–503. https://doi.org/10.1145/3230543.3230582

P4 Language Consortium. 2022. P4 16 language specification. https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html

Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 518–532. https://doi.org/10.1145/3230543.3230548

Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. 2021. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 17–32. https://doi.org/10.1145/3452296.3472937

Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Rudy Peterson, Lennart Beringer, and Andrew W. Appel. 2022. Verifiable P4 : A Foundational Verifier for Stateful P4 Programs. (Sept. 2022). Unpublished manuscript; submitted for publication.

Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 427–440. https://doi.org/10.1145/2103621.2103709

Received 2022-09-21; accepted 2022-11-21