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.

CCS Concepts:
- Software and its engineering → Formal software verification;
- Domain specific languages;
- Semantics;
- Incremental compilers;
- Networks → Programming interfaces.

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

1 Introduction

Well-designed intermediate representations (IR) underpin some of the most successful compiler frameworks including LLVM [18] and CompCert [20]. 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 [1]).

Existing formalizations of P4 are based on the language’s surface syntax, which is complex and unwieldy to work with [8]. 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...
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.

```p4
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_t standard_metadata) {
  state start {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.ethTyp) {
      0x8100: reject;
      0x800: dstAddr = packet.dl_dst;
      dstAddr = packet.dl_src;
    }
    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.

```p4
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.

```p4
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, 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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

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 Cleft, 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 $\langle n \rangle$, 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 $\langle n \rangle$ and int($\langle p \rangle$), with widths of $n$ and $p$ bits respectively. Unlike P4, widths do not need to be multiples of

---

3 Top-level functions were added to the language in version 1.1.0, but with a number of restrictions [22].
Table 1. Metavars.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>𝜏</td>
<td>type</td>
<td>𝑏</td>
<td>bool</td>
</tr>
<tr>
<td>𝑧</td>
<td>integer</td>
<td>𝑛</td>
<td>natural number</td>
</tr>
<tr>
<td>𝑝</td>
<td>positive number</td>
<td>𝑥</td>
<td>string</td>
</tr>
<tr>
<td>𝑒</td>
<td>expression</td>
<td>𝑝𝑟𝑚</td>
<td>parameter</td>
</tr>
<tr>
<td>𝑙</td>
<td>parser state label</td>
<td>𝑝𝑡</td>
<td>parser transition</td>
</tr>
<tr>
<td>𝑠</td>
<td>statement</td>
<td>𝑐𝑑</td>
<td>control declaration</td>
</tr>
<tr>
<td>𝑡𝑑</td>
<td>top declaration</td>
<td>𝑣</td>
<td>value</td>
</tr>
<tr>
<td>𝑙𝑜</td>
<td>left-value</td>
<td>𝑎𝑟𝑔_焱</td>
<td>evaluated argument</td>
</tr>
<tr>
<td>𝑐𝑥</td>
<td>syntactic context</td>
<td>𝜀</td>
<td>typing signals</td>
</tr>
<tr>
<td>𝑒_ณ</td>
<td>value environment</td>
<td>𝜉</td>
<td>extern state</td>
</tr>
<tr>
<td>𝑠𝑔_ณ</td>
<td>evaluation signals</td>
<td>𝜉_焱</td>
<td>value environment</td>
</tr>
<tr>
<td>𝑓𝑛𝑠_焱</td>
<td>function types</td>
<td>𝑓𝑛𝑠</td>
<td>functions</td>
</tr>
<tr>
<td>𝑖𝑛𝑠_焱</td>
<td>instance type</td>
<td>𝑖𝑛𝑠_焱</td>
<td>instance types</td>
</tr>
<tr>
<td>𝑖𝑛𝑠</td>
<td>instance</td>
<td>𝑖𝑛𝑠</td>
<td>instances</td>
</tr>
</tbody>
</table>

8. Since headers are similar to structs, we represent both by struct, 𝑇, where 𝑏 is true for headers and 𝑓𝑎𝑙𝑠𝑒 for standard structs and 𝑇 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.

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

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.
Arguments:

\[
\begin{align*}
\text{arg} & \ ::= \text{in } e \quad \text{in-arguments} \\
& \quad | \quad \text{out } e \quad \text{out-arguments} \\
& \quad | \quad \text{inout } e \quad \text{inout-arguments}
\end{align*}
\]

Parameters:

\[
\begin{align*}
\text{prm} & \ ::= \text{in } \tau \quad \text{in-parameters} \\
& \quad | \quad \text{out } \tau \quad \text{out-parameters} \\
& \quad | \quad \text{inout } \tau \quad \text{inout-parameters}
\end{align*}
\]

Figure 2. P4Cub arguments and parameters.

Statements:

\[
\begin{align*}
s & \ ::= \text{skip} \\
& \quad | \quad \text{return } e \\
& \quad | \quad \text{exit} \\
& \quad | \quad \text{goto } pt \quad \text{parser transition} \\
& \quad | \quad e \ ::= e \quad \text{assignment} \\
& \quad | \quad e \times \mathcal{T} (\text{arg}) \quad \text{function call} \\
& \quad | \quad x \times \mathcal{C} (\text{arg}) \quad \text{action call} \\
& \quad | \quad e \times x (\mathcal{T}) (\text{arg}) \quad \text{method call} \\
& \quad | \quad \text{invoke } x \\
& \quad | \quad \text{apply } x (\text{arg}) \quad \text{apply statements} \\
& \quad | \quad \text{let } e \text{ in } s \quad \text{let binding} \\
& \quad | \quad \text{if } e \text{ then } s \text{ else } s \quad \text{conditional}
\end{align*}
\]

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 external 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 { ... }) {
  action drop()() { mark_to_drop(2); }
  action fwd(bit<9>)( ) { 3.1 = 0; }
  table sw {
    key e { 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 \( \Gamma \) and stores \( \epsilon \) 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 \( \Gamma \) 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 \( \tau := \epsilon \) to indicate appending \( \tau \) to the “beginning” of the environment.

The function \( \text{fns} \) maps a function’s name to the number of type parameters, expression parameters, and return type.
Expression Typing Environment:
\[ \Gamma ::= \bar{\tau} \quad \text{A list of types} \]

Typing Function Environment:
\[ \text{fns}_t ::= \emptyset \quad \text{Empty} \]
\[ \text{fns}_t, x \mapsto (n, \bar{\tau}, \tau) \quad \text{Signature} \]

Instance Types:
\[ \text{inst}_t ::= \bar{\tau}, \bar{\tau} \quad \text{Action signature} \]
\[ (x, n, \bar{\tau}, \tau) \quad \text{External type} \]
\[ \text{Prsr} \bar{\tau} \quad \text{Parser type} \]
\[ \text{Ctrl} \bar{\tau} \quad \text{Control type} \]
\[ \text{Table} \quad \text{Table} \]

Typing Environment:
\[ \text{insts}_t ::= \emptyset \]
\[ \text{insts}_t, x \mapsto \text{inst}_t \]

Typing Syntactic Contexts:
\[ \text{cx} ::= \text{Prsr} n \text{inst}_t \quad \text{Parser} \]
\[ \text{Ctrl} \text{insts}_t \quad \text{Control} \]
\[ \text{Fn} \tau \quad \text{Function} \]

Typing Signal:
\[ \text{sig} ::= \text{Cn} \quad \text{continue} \]
\[ \text{Ex} \quad \text{exit} \]
\[ \text{Rt} \tau \quad \text{return a type} \]
\[ \text{Tr} \quad \text{transition} \]

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

Expression Evaluation Store:
\[ \epsilon ::= \bar{v} \quad \text{A list of values} \]

Evaluation Function Environment:
\[ \text{fns} ::= \emptyset \quad \text{Empty} \]
\[ \text{fns}, x \mapsto (\text{fns}, s) \quad \text{Closure} \]

Evaluation Instances:
\[ \text{inst} ::= \epsilon, \text{insts}, s \quad \text{Action closure} \]
\[ \text{fns}, \text{insts}, s \quad \text{Parser} \]
\[ n, \bar{v}, (x, \bar{\tau}) \quad \text{Table} \]

Evaluation Environment:
\[ \text{insts} ::= \emptyset \]
\[ \text{insts}, x \mapsto \text{inst} \]

Evaluation Syntactic Contexts:
\[ \text{cx} ::= \text{Prsr} n \text{inst}_t \quad \text{Parser} \]
\[ \text{Ctrl} \text{insts}_t \quad \text{Control} \]
\[ \text{Function} \quad \text{Function} \]

Evaluation Signal:
\[ \text{sgl} ::= \text{Cn} \quad \text{continue} \]
\[ \text{Ex} \quad \text{exit} \]
\[ \text{Rt} \quad \text{return a value} \]
\[ \text{Accept} \quad \text{accept} \]
\[ \text{Reject} \quad \text{reject} \]

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 \( \text{insts}_t \) which maps names to instance types. In the implementation they are kept in separate namespaces.

The function \( \text{fns} \) maps a function’s name to the available functions in scope and its body. The instance types \( \text{inst}_t \) have corresponding instances \( \text{inst} \) 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 \( \psi \).

Again for expository simplicity, all “instances” are in the same environment but in the implementation have separate namespaces.

The typing syntactic context \( \text{cx} \) 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 \( \text{Ctrl insts}_t \) 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 \( \text{Prsr n insts}_t \) 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 \( \text{Fn} \tau \) is used when inside of a top-level defined function with the return type \( \tau \). Unlike other contexts, it does not provide any information about...
A typing or evaluation signal, sig and sgl, respectively, indicates whether control flow continues. They are also used to check that a statement is properly formed within its context. Signals such as Cn and Rt are essentially the same as in other imperative languages. An Ex signal indicates that the entire program should stop evaluating. Indeed, Ex halts execution all the way up to the packet-processing pipeline level, whereas Rt only interrupts the enclosing statement block. The Rt τ typing signal returns a type while the evaluation Rt v signal returns a value. Tr 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 Γ n denotes looking up the nth 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 τ n. In T-BinOp, the helper function bop_type ⊕ τ1 τ2 determines the type of the expression based on the binary operator and its operands. As an example, bop_type + bit(n) bit(n) = bit(n). We also take advantage of P4’s numeric data types such as bit(n), which permit one to specify unsigned integers bound by 2n. As an example, T-Index allows any term of type bit(n) 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 [τ] just types its elements, that is, Γ ⊢ τ : [τ].

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, Γ τ p pt τ represents the judgment form of parser transition typing. The rules use some helper functions and predicates. The T-FnCall rule states that calling the function x with the return expression e, τ, type arguments τarg, and arguments τarg results in a Cn signal if x exists in the context and it has τarg type parameters, the return type τr, and parameters τ; the return expression e can be evaluated to an l-value (denoted by helper predicate lvalue_ok e), and e, τarg, and τ type check. To type check e, and τarg type substitutions must be performed using the type arguments τarg. eτ is typed as return type τr substituted with type arguments τarg (denoted by the helper function tsb τarg τr). The arguments τarg are typed as the parameters τ substituted with type arguments τarg (again, denoted by the helper tsb τarg τ). tsb τ τ substitutes de Bruijn type variables in τ with τ; 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, Γ τarg τarg : τ states that arguments τarg have the type τ.

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 lv-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 e 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, (e, τ) || [τ].

Figure 9 shows the big-step semantics of statements. For evaluating statements we need a store e, a context cx, and an external 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:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Constraints</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e : \tau$</td>
<td>$\Gamma \vdash b : \text{bool}$</td>
<td>$\text{T-Bool}$</td>
</tr>
<tr>
<td>$\Gamma n = \tau$</td>
<td>$0 \leq z &lt; 2^n$</td>
<td>$\text{T-Num}$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau n : \tau$</td>
<td>$\Gamma \vdash e.n : \tau$</td>
<td>$\text{T-Mem}$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau r' : \tau'$</td>
<td>$\Gamma \vdash e : \tau$</td>
<td>$\text{T-Meta}$</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \tau'$</td>
<td>$\text{proper_cast} \tau r'$</td>
<td>$\text{T-Idx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash \tau e : \tau$</td>
<td>$\Gamma \vdash e : \tau$</td>
<td>$\text{T-List}$</td>
</tr>
</tbody>
</table>

P4Cub's statement typing rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Constraints</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, \text{fns}, \text{cx} \vdash \text{skip} \rightarrow \text{Cn}$</td>
<td>$\Gamma, \text{fns}, \text{cx} \vdash \text{exit} \rightarrow \text{Ex}$</td>
<td>$\text{T-Skip}$</td>
</tr>
<tr>
<td>$\Gamma, \text{fns}, \text{cx} \vdash \text{lvalue} \rightarrow \text{Cn}$</td>
<td>$\Gamma, \text{fns}, \text{cx} \vdash \text{exit} \rightarrow \text{Ex}$</td>
<td>$\text{T-Ext}$</td>
</tr>
<tr>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\text{T-Cctor}$</td>
</tr>
<tr>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\text{T-List}$</td>
</tr>
<tr>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\Gamma, \text{fns}, \text{cx} \vdash e : \tau$</td>
<td>$\text{T-List}$</td>
</tr>
</tbody>
</table>

environment $\psi$, all other statement evaluation rules simply propagate such a change.

The left-hand side of an assignment in the E-Asgn 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 $\epsilon$ that can be used to update a value in $\epsilon$ 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 $\epsilon$ updates the first element of the array to be true which is

Figure 6. P4Cub expression and statement typing.
sitting at the fifth position in $e$, all other elements of the array remain the same. The E-Asgn rule states that after $e_2$ is fully evaluated to $v$ and $e_1$ is evaluated to an l-value $l_0$, the location represented by $l_0$ in $e$ 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 $e'$ is used to update the call environment $e$ 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 $e'$, and assigns 1 to false in $e$. Arguments specified as inout serve as both in and out. E-MtdCall also uses copy_in and copy_out. Because external methods are externally defined, not in the program syntax, E-MtdCall must make use of $\psi$ and execExtern 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, $\text{insts} x = (e_{ct}, \text{insts}', s)$. 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 $e_1 \ast e_2$, and the next parser state is then evaluated using $e_2$ 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 $|e_2|$ parameters/arguments. Thus when transitioning states in E-TransI only the last $|e_2|$ values in the environment $e_1 \ast e_2$ should be used when evaluating the next state: $e_1$ represents variables introduced within the current parser block before the transition takes place. Similarly, in E-Invoke a list append $e_1 \ast e_2$ is used to separate the values in the environment. $e_2$ is the part of the environment with de Bruijn indices in scope.
Figure 9. P4Cub statement evaluation.
at the table’s definition. $e_i$ 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 $e_1 \neq e_2$, a suffix $e_2$ 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 $|e_i|$ 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 [8] 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 simplifies the effort.

6.1 Metatheory

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

**Theorem 7.** Expression evaluation preserves typing.

$$\forall e \in \Gamma \vdash e : \tau \rightarrow e : \tau \rightarrow \exists v, \langle e, v \rangle \vdash \tau$$

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

$$\forall \Gamma \vdash e : \tau \rightarrow e : \tau \rightarrow \exists v, \langle e, v \rangle \vdash \tau$$

Furthermore, we have shown preservation and progress 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 $e : \Gamma$, which indicates that all of the values in the store $e$ have type $\Gamma$ 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 $\Gamma$ and evaluation $e$ as opposed to having mappings from strings to types or values. $e : \Gamma$ succinctly indicates both that $\Gamma$ and $e$ have the same domain of (de Bruijn) variable names and that their elements type correspondingly.

Expression evaluation is also deterministic:

**Theorem 9.** Determinism.

$$\forall e \in \Gamma \vdash e : \tau \rightarrow e : \tau \rightarrow \exists v, \langle e, v \rangle \vdash \tau$$

In Coq it looks like:

Theorem expr_deterministic : forall e v1 v2, (e, e) v1 -> (e, e) v2 -> v1 = v2.

Proof.

intros eps 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_impl_Formal2

- - - - - - - 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.

$$\forall fns \psi' e' c s sgl, \langle \psi, fns, c, s \rangle \vdash e' \rightarrow |e| = |e'|$$

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 $\Gamma$. This proof has been automated in Coq.

Lemma sbs_length : forall es e' c s sig sgl, (forall es e' c s sig sgl) -> length e = length e'.

Proof using.


induction h; autorewrite with core in *; 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 [10], 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($n$) and int($p$) 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 \( z(n) \) must be moved to the statement level in-order to be compiled to Clicht.

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 \( (e \mapsto \bar{e}, e') \), works by generating both a new “lifted” term, \( e' \), as well as list of terms that will become variable declarations, \( \bar{e} \). 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 \( \text{eval} \_\text{decl} \_\text{list} \ e \ \bar{v} \), which says that in context \( e, \bar{v} \) evaluates to \( \bar{v} \).

**Theorem 11.**
\[
\forall e \ e' \ \bar{e} \ \bar{v}, (e, e') \Downarrow v \rightarrow e \mapsto \bar{e}, e' \rightarrow \bar{v} \square, \text{eval} \_\text{decl} \_\text{list} \ e \ \bar{v} \\
\land (\Downarrow \bar{v} \Downarrow e, e') \Downarrow v
\]

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 \( \bar{v} \) in the specification is expressed as \( \text{eval} \_\text{decl} \_\text{list} \ e \ \bar{v} \), 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 \( p4v \) [21], Aquila [25], and Vera [24]. The core of this verifier is a compiler from P4Cub to Dijkstra’s Guarded Command Logic (GCL) [7]. 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 (external 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 [22] requires parser loops to be finitely unrollable. So we can get away with providing an additional \( \text{unroll} \) parameter that specifies how many times to unroll the parser. There’s a subtle difference between the \( \text{unroll} \) and gas parameters—running gas to \( \emptyset \) triggers a compilation failure, prompting the user to try again with more, while running \( \text{unroll} \) to \( \emptyset \) 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 [25], 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 [25]. Each state \( s \) (including \( \text{accept} \) and \( \text{reject} \) is given a 1-bit ghost variable \( \_\text{state} \_\text{&} \_\text{statenext} \), 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 \( p4v \) [21].

Retaining tables in P4Cub leaves verifiers freedom in modeling tables. Tables can be compiled away using ghost variables [21]. However, different ghost variable models are appropriate for different verification tasks [6, 12, 24]. The choice of table model affects compilation and verification
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.

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 [7, 13], 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. The so-called header validity problem is akin to the pointer nullability problem in Java or C, and has been heavily studied [5, 11, 12, 21, 24].

7 Related Work

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

Intermediate Representations. LLVM [18] is perhaps the most well-known modern compiler IR—it’s SSA abstraction allows for efficient compilation of many languages. One of the more notable success stories is the Clang compiler [17] for C/C++ and Objective-C. The MLIR project [19] 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 [15], 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 [20], 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 [27] provides a formal semantics for a subset of LLVM, to facilitate the development of certified LLVM compilers. Finally, theVerified Software Toolchain [3] is an ongoing project developing static analyzers, program verifiers, and compilers for the C programming language, including program logics like Verifiable C [4].

P4 Verification. Petr4 [8] and P4K [16] 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 [11, 12], though none suffice as compiler IRs, being themselves highly idealized versions of the language. P4v [21], Aquila [25], Vera [24], and Assert-P4 [14] are P4 program verifiers that translate P4 to GCL to compute verification conditions. The closest lines of work to P4Cub are Verifiable P4 [26] and HOL-P4 [2]. Verifiable P4 is a program logic for proving properties of 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.

9 Data-Availability Statement
This paper’s artifact is available here [23]. A full release is available here [9].

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:

\[
\begin{aligned}
\text{cd} & ::= \text{var e} & \text{local} \\
& | \text{action } x(\overline{\tau}(\overline{prm}) \{s\}) & \text{action} \\
& | \text{table } x \{\text{key } = \overline{\tau}, \text{actions } = \{\overline{x}, \overline{arg}\}\} & \text{table}
\end{aligned}
\]

Figure 11. P4Cub declarations within controls.

Top-level declarations:

\[
\begin{aligned}
\text{td} & ::= \text{instance } x \text{ of } x(\overline{\tau}) & \text{instantiate} \\
& | \text{extrn } x(n)(\overline{\tau}(\overline{n})(\overline{prm}) & \text{extern} \\
& | \text{ctrl } x(\overline{prm})\{cd\} & \text{controls} \\
& | \text{prsr } x(\overline{prm}) & \text{parsers} \\
& \text{start } = s \{s\} & \text{functions}
\end{aligned}
\]

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 \(\overline{\tau}\), 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 \(\{x, \overline{arg}\}\) names actions to call. Each action name is paired with data plane arguments provided by the programmer.

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.
Parser transition typing:

\[
\begin{align*}
\text{valid_state } n & \vdash_p \text{pt} \\
\end{align*}
\]

\begin{align*}
\text{T-DirectTrans} & : \Gamma \vdash_p n \quad \text{direct } n \\
\text{T-SelectTrans} & : \Gamma \vdash_p \text{select } e \{ \text{pat } \Rightarrow l \}
\end{align*}

Pattern typing:

\begin{align*}
\text{pat} : \tau & \Rightarrow n, \Gamma \vdash_p \text{pat} \\
\text{T-Wild} & : \Gamma \vdash_p \text{pat} \\
\text{T-IntPat} & : \Gamma \vdash_p \text{pat}_1 \&\& \text{pat}_2 : \text{bit}(n) \\
\text{T-Range} & : \Gamma \vdash_p \text{pat}_1 .. \text{pat}_2 : \text{bit}(n) \\
\text{T-ListPat} & : \Gamma \vdash_p \text{pat} : \tau
\end{align*}

Argument typing:

\begin{align*}
\Gamma \vdash_{\text{arg}} \text{arg} : \tau
\end{align*}

\begin{align*}
\Gamma \vdash_e \text{e} : \tau & \Rightarrow \Gamma \vdash_{\text{arg}} \text{arg} : \tau \\
\text{T-In} & : \Gamma \vdash_{\text{arg}} \text{in} : \tau \\
\text{T-Out} & : \Gamma \vdash_{\text{arg}} \text{out} : \tau \\
\text{T-INOUT} & : \Gamma \vdash_{\text{arg}} \text{out} : \tau
\end{align*}

Figure 13. Helper judgments for P4Cub statement typing.

Figure 14. Helper judgments for statement evaluation.

Parser transition expression evaluation:

\begin{align*}
\langle e, pt \rangle \parallel_p l & \Rightarrow \langle e, \text{out } e \rangle \parallel_p \text{out } l \\
\langle e, \text{direct } l \rangle \parallel_p l & \Rightarrow \langle e, \text{select } e \{ \text{pat } \Rightarrow l' \} \rangle \parallel_p l' \\
\langle e, \text{direct } l \rangle \parallel_p l & \Rightarrow \langle e, \text{select } e \{ \text{pat } \Rightarrow l' \} \rangle \parallel_p l'
\end{align*}

L-value evaluation:

\begin{align*}
\langle e, \text{in } e \rangle \parallel_p \text{in } e & \Rightarrow \langle e, \text{out } e \rangle \parallel_p \text{out } l \\
\langle e, \text{in } e \rangle \parallel_p \text{in } e & \Rightarrow \langle e, \text{out } e \rangle \parallel_p \text{out } l
\end{align*}

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. 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.
whether to evaluate an argument to a value or a l-value. metadata from headers is done by externs, and is opaque in Note that in P4Cub, as in P4, the actual work of extracting evaluation simply evaluates the parser transition expression. The parser transition arguments, shown in Figure 9. The parser transition evaluation judgment determines the type of an argument.

Finally, the argument typing judgment determines whether 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.

```
\[
\begin{array}{ll}
e \doteq \overline{e}, \overline{e}' \\
\end{array}
\]
```

\[
\begin{array}{ll}
\overline{e} \doteq \overline{\overline{e}} \\
\end{array}
\]

\[
\begin{array}{ll}
\overline{\overline{\overline{e}}} & \doteq \overline{\overline{\overline{\overline{e}}}} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{eval_decl_list } e \overline{\overline{\overline{e}}} & \doteq \overline{\overline{\overline{\overline{e}}}} \\
\end{array}
\]

\[
\begin{array}{ll}
\langle \overline{\overline{\overline{e}}} \rangle \doteq (\overline{\overline{\overline{\overline{e}}}}) \doteq \overline{\overline{\overline{\overline{e}}}} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{Figure 16. Evaluation of lifted list.} \\
\end{array}
\]

This is needed because inter-procedural calls in P4 such as those to functions and actions have a copy-in and copy-out semantics.

### References


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