# Petr4: Formal Foundations for P4 Data Planes

RYAN DOENGES, Cornell University, USA
MINA TAHMASBI ARASHLOO, Cornell University, USA
SANTIAGO BAUTISTA*, ENS Rennes, France
ALEXANDER CHANG, Cornell University, USA
NEWTON NI, Cornell University, USA
SAMWISE PARKINSON, Cornell University, USA
RUDY PETERSON, Cornell University, USA
ALAIA SOLKO-BRESLIN, Cornell University, USA
AMANDA XU, Cornell University, USA
NATE FOSTER, Cornell University, USA

P4 is a domain-specific language for programming and specifying packet-processing systems. It is based on an elegant design with high-level abstractions like parsers and match-action pipelines that can be compiled to efficient implementations in software or hardware. Unfortunately, like many industrial languages, P4 has developed without a formal foundation. The P4 Language Specification is a 160-page document with a mixture of informal prose, graphical diagrams, and pseudocode, leaving many aspects of the language semantics up to individual compilation targets. The P4 reference implementation is a complex system, running to over 40KLoC of C++ code, with support for only a few targets. Clearly neither of these artifacts is suitable for formal reasoning about P4 in general.

This paper presents a new framework, called Petr4, that puts P4 on a solid foundation. Petr4 consists of a clean-slate definitional interpreter and a core calculus that models a fragment of P4. Petr4 is not tied to any particular target: the interpreter is parameterized over an interface that collects features delegated to targets in one place, while the core calculus overapproximates target-specific behaviors using non-determinism.

We have validated the interpreter against a suite of over 750 tests from the P4 reference implementation, exercising our target interface with tests for different targets. We validated the core calculus with a proof of type-preserving termination. While developing Petr4, we reported dozens of bugs in the language specification and the reference implementation, many of which have been fixed.

CCS Concepts: • **Software and its engineering → Formal language definitions**; • **Networks → Programming interfaces**.

Additional Key Words and Phrases: P4, formal semantics

**ACM Reference Format:**
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. *Proc. ACM Program. Lang.* 5, POPL, Article 41 (January 2021), 32 pages. https://doi.org/10.1145/3434322

*Work performed at Cornell University.

Authors' addresses: Ryan Doenges, Cornell University, Ithaca, NY, USA, rhd89@cornell.edu; Mina Tahmasbi Arashloo, Cornell University, Ithaca, NY, USA, mt822@cornell.edu; Santiago Bautista, ENS Rennes, Bruz, France, santiago.bautista@ens-rennes.fr; Alexander Chang, Cornell University, Ithaca, NY, USA, apc73@cornell.edu; Newton Ni, Cornell University, Ithaca, NY, USA, cn279@cornell.edu; Samwise Parkinson, Cornell University, Ithaca, NY, USA, stp59@cornell.edu; Rudy Peterson, Cornell University, Ithaca, NY, USA, rnp39@cornell.edu; Alaia Solko-Breslin, Cornell University, Ithaca, NY, USA, ajs644@cornell.edu; Amanda Xu, Cornell University, Ithaca, NY, USA, ax49@cornell.edu; Nate Foster, Cornell University, Ithaca, NY, USA, jnfoster@cs.cornell.edu.

## 1 INTRODUCTION

Most networks today are designed and operated without the use of formal methods. The philosophy of the Internet Engineering Task Force (IETF), which manages the standards for protocols like TCP and IP, can be summarized by David Clark's slogan: "we believe in rough consensus and running code." Likewise, Jon Postel's famous dictum to "be conservative in what you do, be liberal in what you accept from others," advocates for a kind of robustness that is achieved not by adhering to precise logical specifications, but rather by designing systems that can tolerate minor deviations from perfect behavior.

But while it is hard to argue with the success of modern networks, one only has to glance at the recent headlines to see that operating a network correctly is becoming a huge challenge, especially at scale [Svaldi 2019]. Hardware and software bugs frequently rear their heads, causing service outages, performance degradations, and security incidents.

Given this context, it is natural to ask whether formal methods may assist in building networks that behave as intended. Indeed, a number of recent tools including Header Space Analysis (HSA) [Kazemian et al. 2012], Anteater [Mai et al. 2011], NetKAT [Anderson et al. 2014], Batfish [Fogel et al. 2015], Minesweeper [Becket et al. 2017], ARC [Gember-Jacobson et al. 2016], and others enable operators to automatically verify a variety of network-wide properties. Startup companies like Forward Networks, Veriflow Systems, and Intentionet offer commercial products based on these tools, and even large companies like Amazon [Dodge and Quigg 2018], Cisco [Cisco Systems 2018], and Microsoft [Bjorner and Jayaraman 2015; Liu et al. 2017] have invested in network verification.

Despite significant progress, there is a widening gap between the simple models used by network verification tools, and the growing set of features supported on modern routers and switches. Early tools like HSA and VeriFlow were based on OpenFlow, a stateless packet-forwarding model that handles about a dozen basic protocols. However, today a typical data center switch supports 40 or more conventional protocols (e.g., Ethernet, ARP, VLAN, IPv4, TCP, and UDP), and new protocols (e.g., VXLAN, Segment Routing, and ILA) are rapidly emerging. Moreover, even when the protocols are well understood, it can be difficult to collect the inputs that verification tools require because device configurations are usually written in idiosyncratic, vendor-specific formats.

*P4 language.* A promising idea for addressing these challenges is to encode the behavior of each device in a common representation that is amenable to analysis. In particular, the P4 language [Bosshart et al. 2014; The P4 Language Consortium 2018] provides a collection of domain-specific abstractions (e.g., header types, packet parsers, and match-action tables) that can be used to describe the functionality of a wide range of packet-processing systems. At Google, P4 is used as a specification language for fuzzing fixed-function switches [Heule et al. 2019], but the language is flexible enough to implement or specify completely new forwarding behavior—e.g., in-band telemetry [Hira and Wobker 2015] or in-network computing [Jin et al. 2018, 2017] techniques.

Unfortunately, although P4 has been gaining momentum in industry as both an implementation and specification language, it lacks a solid semantic foundation. The official definition of P4 is an informal document maintained by a language design committee. It describes operational behavior in pseudocode and does not give a complete definition of a type system, so it is not always clear what a given program construct means. Turning to the open-source reference implementation of P4 does not provide clear guidance either, because it is complex, contains bugs, and occasionally diverges from the specification. Besides hindering our understanding of P4 program behavior, the absence of a clear formal foundation for P4 has made it difficult to understand and evolve the language itself. For instance, bounded polymorphism has been a topic of discussion in the language design committee for over three years, but without the type system written down it is difficult to see how such an extension would interact with existing language features.

*The Petr4 framework.* This paper presents Petr4 (pronounced "petra"—i.e., Greek for stone), a new framework that puts P4[1] on a solid foundation. Petr4 is based on two distinct contributions: (i) a clean-slate definitional interpreter for P4, and (ii) a core calculus modeling a fragment of P4. The two artifacts were designed to be consistent—we developed the calculus after building the interpreter—but they are not formally related. Our implementation offers a front-end, type checker, interpreter, and test harness, as well as command-line and web-based user interfaces. Our calculus defines the meaning of simple P4 programs using standard typing and evaluation judgments.

Petr4 builds on standard techniques developed by the programming languages community over several decades and applies these tools to a large, industrial language in a new domain. In building Petr4 we had to overcome several challenges. First, as has already been mentioned, the official definition of P4 is a 160-page specification document containing informal prose, graphical diagrams, snippets of code, and a grammar. But while the document is generally well written, there are some surprising inconsistencies and omissions—e.g., it does not define P4's lexical syntax or its type system precisely. Second, P4 is a low-level language with a variety of constructs for bit-level operations. There are subtle issues that arise with undefined values, casts, and exceptional control flow that require a careful treatment. Third, P4 is not really a single language but a family of languages—there is one dialect for each architecture that it supports. Hence, to fully understand the meaning of a P4 program, one must also understand the semantics of its intended target.

To address these challenges, we first studied the language specification, reporting dozens of bugs, ambiguities, and inconsistencies to the language design committee. We then built a clean-slate definitional interpreter, carefully following the specification rather than adapting code from the open-source implementation (i.e., to avoid replicating bugs). One unusual aspect of our interpreter is that it is parameterized on the choices delegated to architectures—e.g., what happens when reading or writing an invalid packet header. We developed a "plug-in" model that allows the interpreter to be instantiated for new architectures, often in only a few hundred lines of OCaml. We validated our semantics against the test suite for the open-source implementation, which uncovered additional bugs. Finally, we extracted a core calculus from our implementation and proved key properties including termination, using nondeterminism to account for target-specific operational behavior.

*Contributions.* Overall, this paper makes the following contributions:

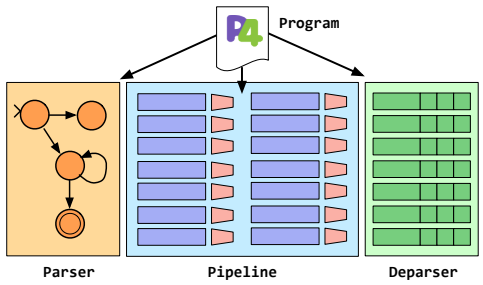
- We develop a clean-slate, definitional interpreter for the P4 language (§ 4).
- We define a calculus that models the semantics of a core fragment of P4 in terms of standard typing and operational semantics judgments. (§ 3).
- We prove type soundness and termination (§ 3) for our calculus.
- We develop an extension to the language (§ 6) as a case study.
- We validate our implementation against hundreds of tests from the test suite for P4's reference compiler and classify some of the bugs we found (§ 5).

Overall, Petr4 represents a promising first step toward the vision of formally verified systems built using P4. In particular, we are optimistic that Petr4 will not only provide a rigorous foundation for current language-based verification tools, but will also serve as a catalyst for future efforts that target higher layers of the networking stack.

## 2 BACKGROUND

This section introduces the P4 language using a simple example and motivates the need for formal foundations by highlighting some of the opportunities and challenges related to formal reasoning about P4 programs.

[1] In this paper, when we refer to P4, we mean $P4_{16}$, and not earlier versions of the language.




(a)

```
// Architecture externs for packet I/O
extern packet_in {
  void extract<T>(out T hdr);
}
extern packet_out {
  void emit<T>(in T hdr);
}

// Architecture metadata
struct std_meta {
  // metadata initialized on ingress
  bit<8> ingress_port;
  bit<32> packet_length;
  // metadata controlling egress
  bit<8> egress_port;
}

// Architecture types
parser Parse<H>(packet_in pkt,
                out H hdrs,
                inout std_meta meta);
control Pipeline<H>(inout H hdrs,
                    inout std_meta meta);
control Deparse<H>(packet_out pkt,
                   in H hdrs);

// Architecture package
package Switch<H>(Parse<H> parse,
                  Pipeline <H> pipe,
                  Deparse<H> deparse);
```

(b)

```
// Programmer-defined types
header hop {
  bit<7> port;
  bit<1> bos;
}
struct headers {
  hop[9] hops;
}
// Programmer-defined components
parser MyParse(packet_in pkt,
               out headers hdrs,
               inout std_meta meta) {
  state start {
    pkt.extract(hdrs.hops.next);
    transition select(hdrs.hops.last.bos) {
      1: accept;
      default: start;
    }
  }
}
control MyPipe(inout headers hdrs,
               inout std_meta meta) {
  action allow() { }
  action deny() { meta.egress_port = 0xFF; }
  table acl {
    key = { meta.ingress_port : exact;
            meta.egress_port : exact; }
    actions = { allow; deny; }
    default_action = deny();
  }
  apply {
    meta.egress_port =
      (bit<8>)hdrs.hops[0].port;
    hdrs.hops.pop_front(1);
    acl.apply();
  }
}
control MyDeparse(packet_out pkt,
                  in headers hdrs) {
  apply { pkt.emit(hdrs.hops); }
}
Switch(MyParse(),MyPipe(),MyDeparse()) main;
```

(c)

Fig. 1. Example: (a) Diagram of three-stage architecture; (b) P4 definition of three-stage architecture; (c) simple P4 program that implements source routing with access control in the three-stage architecture.

*Targets and architectures.* P4 is a domain-specific language designed for programming a range of packet-processing targets, including high-speed routers, software switches, and network interface cards (NICs). Although the details of these targets vary, they tend to have a few features in common, including a programmable parser that maps input packets into typed representations for processing and a pipeline that interleaves reconfigurable tables and fixed-function blocks. Some targets offer limited forms of persistent state that can be read and written by each packet, but they typically do not support general recursion—looping would require sending the packet through the pipeline multiple times, which degrades throughput.

The core constructs in P4 capture what these targets have in common. To accommodate their differences, it also provides the notion of an architecture, which exposes the structure and capabilities of the underlying target while abstracting away implementation details. For example, Figure 1 (a) depicts the structure of a simple architecture that processes packets in three stages: the input packet is first parsed into a typed representation using a finite-state machine, then the parsed representation is transformed using a sequence of match-action tables and arithmetic units, and finally the parsed representation is serialized into the output packet.

*Example program: three-stage architecture.* Figure 1 (b) shows how this three-stage architecture can be defined as a part of a P4 program. The architecture definitions should be read like a Java interface or ML module signature—they specify the structure and types of each component, but do not define their implementation. The first few declarations define the types of `extern` objects that can be used to map between raw packets and typed representations. For instance, the `packet_in` object's `extract` method reads from the input packet and populates the header passed as an argument. The next few declarations define a `struct` type for the metadata associated with the architecture, including the `ingress_port`, which is initialized by the target when a packet is received; and the `egress_port`, which specifies the port to use when emitting the packet. The last few declarations define the P4-programmable components of the architecture: a parser, a pipeline, and a deparser, as well as the `package` that models the device itself.

*Example program: source routing with access control.* Figure 1 (c) defines a P4 program written against the three-stage architecture that implements a simple form of source routing and access control. Here source routing means that each packet carries a stack of values that encodes the series of ports the packet should be forwarded out on as it traverses the network, while access control means the control plane can install filtering rules in a match-action table to drop certain packets. More formally, each packet has a fixed-length array (or "stack") of byte-sized `hop` headers. Each header is initially "invalid" but becomes "valid" when it is populated by the parser. For the `hop` header, the first 7 bits encode the output port and the 8th "bottom of stack" bit is 1 if it is the last element in the stack. The `MyParse` parser uses a finite state machine abstraction to map raw input packets into this typed representation. The parser has a single state that repeatedly extracts `hop` headers from the packet until the `bos` ("bottom of stack") marker is 1. Note that because packets are finite and the loop extracts some bits from the packet on each iteration, the parser is guaranteed to terminate. Next, the `MyPipe` control defines an `apply` method that specifies how packets are processed. This method sets the `egress_port` metadata field to the port encoded in the top element of the stack, pops the stack, and then executes the `acl` table, which matches the `ingress_port` and `egress_port` metadata fields against filtering rules (not shown) installed by the control-plane. The rules either `allow` or `deny` the packet, defaulting to `deny` if no matching rule can be found. Finally, the `MyDeparse` control serializes the parsed data back into an output packet.

*Formal methods opportunities and challenges.* At first glance, P4 appears to be a relatively simple language. So it seems like it should be possible to use P4 to reason formally about a range of network scenarios, such as the following:

- *Executable specifications of protocols:* Rather than specifying protocols using informal documents, like IETF RFCs, we could use P4 to create executable protocol specifications that precisely specify packet formats and allowed behaviors. For example, the program in Figure 1 (c) might serve as the definition of the source routing scheme it realizes. Whereas current efforts to standardize protocols rely on informal ASCII documents, the P4 program would provide an unambiguous, mechanized, executable reference that could be used to design and validate other implementations.

- *Program verification:* P4 programs are expected to satisfy various properties—e.g., an IPv4 router should correctly decrement the `ttl` field and also unambiguously specify the forwarding behavior of each packet. Generally speaking, verification is simpler than in many other languages because P4 lacks complex data types and iteration. But current P4 verification tools [Liu et al. 2018; Stoenescu et al. 2018] rely on existing front-ends such as the open-source reference implementation, which is known to deviate from the specification and has bugs. Hence, the results of verification are potentially compromised.
- *Verified compilers:* P4 compilers must generate low-level code for hardware devices such as programmable switches and FPGAs. This process transforms the input program in complex ways—e.g., unrolling parser state machines, eliminating common sub-expressions, and extracting parallelism for hardware pipelines. Many of these transformations rely on intricate side conditions that are easy to get wrong [Ruffy et al. 2020]. A verified compiler for P4, either using static verification or translation validation, could eliminate bugs in compilers and make it possible to obtain implementations that are guaranteed to be correct.
- *Proof-carrying code:* Today, cloud platforms allow customers to customize the network infrastructure to suit their needs—e.g., they can obtain an isolated virtual network slice that they can configure however they like. In the near future, cloud providers are likely to go further and allow customers to customize the low-level behavior of devices such as routers and smart NICs. Techniques like proof-carrying authorization and proof-carrying code [Skalka et al. 2019] could be used to allow P4 programs written by different customers to collaborate to implement new features without interfering with the functionality of the network as a whole.

Unfortunately, while these examples represent some exciting applications of formal methods to networks, realizing them today would be difficult. The key challenge is that P4 lacks a formal foundation, so it is difficult to reason about the language and its programs. More specifically, we identify three challenges that any formalization of P4 must overcome:

- *Incomplete specification*: The language specification is generally well-written but does not fully specify the meaning of each language construct. For example, the type system is only described at a high level, and important questions such as the precise semantics of implicit casts and the definition of type equivalence are left unanswered. There are also tricky interactions between features that have apparently never been considered, such as whether extern objects can have recursive types.
- *Undefined values*: To ease compilation to resource-limited targets, P4 makes certain trade-offs between safety and efficiency. For example, P4 allows programs to manipulate uninitialized or invalid headers; reading or writing an invalid header yields an undefined value. For example, the forwarding behavior of the program in Figure 1 (c) is undefined in cases where `hops[0]` is invalid.
- *Architecture-specific behaviors*: The P4 specification also delegates many key decisions to architectures, making the meaning of a P4 program architecture-dependent. To give one example, the behavior of the program in Figure 1 (c) depends on whether malformed packets—i.e., with more than 9 `hops` headers—are automatically dropped by the parser or propagated to the pipeline. Other architecture-specific behaviors and restrictions include the matches and actions supported in match-action tables and the availability of certain arithmetic operations such as division.

To reason precisely about the behavior of a P4 program today, a programmer has two main options: they can consult the language specification or they can execute the program with an existing implementation. Of course, there are serious issues with either choice. The specification is incomplete, and the implementations restrict programs to the behavior of specific targets.

*Our approach.* Our primary goal in developing Petr4 was to produce a reusable, realistic formal semantics for P4. In particular, we wanted to support executing programs in a manner that precisely follows the existing specification (to the extent possible), and facilitate doing formal proofs about programs as well as the language as a whole. To this end, we developed a clean-slate definitional interpreter for P4 in OCaml, and we also designed a calculus that models the type system and operational semantics for a core fragment of the language. Working carefully from the specification, our implementation was designed to be independent of the existing open-source implementation. To resolve situations where the specification was vague or delegated decisions to architectures, we parameterized our development, allowing each target to make a different choice. For example, our calculus models undefined values using an oracle, and our interpreter is an OCaml functor that can be instantiated to realize the behavior of different architectures. Overall, we believe that Petr4 represents a promising first step toward our vision of verified data planes, offering a rigorous foundation as well as running code.

## 3 CORE P4

This section presents the syntax and semantics of Core P4, a simple language that models the essential features of P4 in a core calculus.[2] P4 is a large and idiosyncratic language and while our definitional interpreter handles nearly all of its features, formalizing the full language in a paper would be unwieldy. We offer here a selective transcription of the semantics realized in the Petr4 implementation. The semantics is sufficiently rich to capture the feature interactions that make P4 tricky to reason about, while avoiding the notational clutter of the full language. The most significant omission from Core P4 is parsers. Hence, Core P4 models the essential packet-processing done by `control` blocks but omits recursion, which allows us to prove a termination result.

If desired, parsers that have been unrolled to eliminate recursion can be emulated using Core P4's functions, with one function for each state. This retains the termination theorem and is often done in practice on resource-constrained targets. Indeed, the P4 specification states that compilers "may reject parsers containing loops that cannot be unrolled at compilation time."

### 3.1 Syntax and Examples

Core P4 is a mostly standard imperative language with separate syntactic classes for expressions, statements, and declarations. It includes mutable variables, generic functions, and familiar types like booleans and records. Target-specific functionality is made available with "native" functions. For example, the following Core P4 program models the `apply` block from Figure 1 (c):

```
meta.egress_port := (bit⟨8⟩) hops[0].port;
pop_front(hops, 1);
acl();
```

The Core P4 program uses function calls to model header stack operations (pop_front) and match-action table invocations (acl), but is otherwise identical to the original program.

The P4 specification imposes a multitude of restrictions on type nesting, parameter types, locations of instantiations, and other language constructs. While we stratify the Core P4 type system to prevent higher-order phenomena, we avoid modeling the remainder of the specification's restrictions in Core P4. Nonetheless, Core P4 is type safe. The restrictions aim to simplify compiling P4 programs to sometimes idiosyncratic and resource-limited hardware targets. They are not fundamental and could be lifted by new P4 compilation strategies.

[2]For space, some rules and all proofs are relegated to an appendix available on arXiv [Doenges et al. 2020].

$$
\begin{array}{rcll}
\rho & ::= & \mathsf{bool} & \text{booleans} \\
& | & \mathsf{int} & \text{integers} \\
& | & \mathsf{bit}\langle exp \rangle & \text{bitstrings} \\
& | & \mathsf{error}\ \{\overline{f}\} & \text{errors} \\
& | & \mathsf{match\_kind}\ \{\overline{f}\} & \text{match kinds} \\
& | & \mathsf{enum}\ X\ \{\overline{f}\} & \text{enums} \\
& | & \{\overline{f : \rho}\} & \text{records} \\
& | & \mathsf{header}\ \{\overline{f : \rho}\} & \text{headers} \\
& | & \rho[n] & \text{stacks} \\
& | & X & \text{type variables}
\end{array}
\qquad
\begin{array}{rcll}
\tau & ::= & \rho & \text{data types} \\
& | & \mathsf{table} & \text{tables} \\
& | & \mathsf{function}\langle\overline{X}\rangle(\overline{d\ x : \rho}) \to \rho_{ret} & \text{functions} \\
& | & \mathsf{ctor}(\overline{x : \tau}) \to \tau_{ret} & \text{constructors} \\
\\
d & ::= & \mathsf{in} & \text{copy-in} \\
& | & \mathsf{out} & \text{copy-out} \\
& | & \mathsf{inout} & \text{copy-in-out}
\end{array}
$$

Fig. 2. Core P4 types and directions.

*3.1.1 Notational conventions.* We typeset metavariables in *italics* and keywords and other concrete identifiers in sans serif. We avoid explicit indexing of sequences by writing a line over the term we would otherwise index. For instance, $\overline{x}$ represents a list $x_1, x_2, \ldots, x_n$. We write $x$ for ordinary variables and $X$ for type variables and names. We write $f$ for fields of records or members of enumeration and "open enumeration" types. There are two open enums, which have the reserved type names error and match_kind. Locations $\ell$ appear in the dynamic semantics. We write $\ell$ fresh to obtain a new location $\ell$.

*3.1.2 Types (fig. 2).* Core P4 types are stratified into function types $\tau$ and base types $\rho$, with generics only allowed to range over base types. We often allow the syntax of types to implicitly constrain a type's "level" rather than explicitly writing them as $\rho$. So for example function declaration parameters are always base types, because a function type can only have base types as its arguments, but we write them with a $\tau$ metavariable and allow the level to be inferred from context.

Numeric datatypes in P4 are flexible. Consider this header type representing an IPv4 option:

$$
\begin{array}{l}
\mathsf{header}\ \{\ \mathsf{copyFlag} : \mathsf{bit}\langle 1\rangle \\
\qquad\qquad\ \ \mathsf{optClass} : \mathsf{bit}\langle 2\rangle \\
\qquad\qquad\ \ \mathsf{option} : \mathsf{bit}\langle 5\rangle \\
\qquad\qquad\ \ \mathsf{optionLength} : \mathsf{bit}\langle 8\rangle\ \}
\end{array}
$$

Each field is an unsigned integer with its width specified in angle brackets. This is convenient when describing network protocol wire formats, which do things like pack 1- and 7-bit values into a byte without padding. P4 even allows the width of a type to be an expression, provided it can be evaluated at compile time. The presence of expressions in types complicates type equality, as can be seen in this short example.

$$
\mathsf{const\ int\ w} := 8;\ \mathsf{bit}\langle \mathsf{w}\rangle\ \mathsf{x} := 1;\ \mathsf{bit}\langle 8\rangle\ \mathsf{y} := \mathsf{x};
$$

The type $\mathsf{bit}\langle \mathsf{w}\rangle$ is not syntactically equal to $\mathsf{bit}\langle 8\rangle$, but the type checker should permit the assignment. The Core P4 type system handles this by reducing types to a normal form before comparing the normal forms with syntactic equality (modulo $\alpha$-equivalence for generics). The implementation of this equality check will in some situations impose type equality by inserting casts, but we do not model implicit casts in Core P4.

The match_kind and error types are "open enumerations," comparable to the extensible exception type in Standard ML [Milner et al. 1997]. Repeated declarations extend the open enumeration with new members without replacing the old members or shadowing the existing type.

| | | | | | | |
|---|---|---|---|---|---|---|
| $exp$ | ::= | $b$ | booleans | $stmt$ | ::= | $exp\langle\overline{\rho}\rangle(\overline{exp})$ | method call |
| | \| | $n_w$ | integers | | \| | $exp := exp$ | assignment |
| | \| | $x$ | variables | | \| | if $(exp)$ $stmt$ else $stmt$ | conditional |
| | \| | $exp_1[exp_2]$ | array accesses | | \| | $\{\overline{stmt}\}$ | sequencing |
| | \| | $exp_1[exp_2{:}exp_3]$ | bitstring slices | | \| | exit | exit |
| | \| | $\ominus\, exp$ | unary ops | | \| | return $exp$ | return |
| | \| | $exp_1 \oplus exp_2$ | binary ops | | \| | $var\_decl$ | variable declaration |
| | \| | $(\rho)\ exp$ | casts | $lval$ | ::= | $x$ | local variables |
| | \| | $\{\overline{f = exp}\}$ | records | | \| | $lval.f$ | fields |
| | \| | $exp.f$ | fields | | \| | $lval[n]$ | array elements |
| | \| | $X.f$ | type members | | \| | $lval[n_1 : n_2]$ | bitstring slices |
| | \| | $exp\langle\overline{\rho}\rangle(\overline{exp})$ | function call | | | | |

Fig. 3. Core P4 expression, statement, and l-value syntax. The expression on the left of an assignment is not an l-value to allow (for example) a computed array index, which evaluates to an l-value with a fixed index.

*3.1.3 Expressions (fig. 3).* Core P4 offers a rich set of expressions for manipulating packet contents. For example, the following program extracts the 6th byte of a bitstring bits:

$$\begin{aligned}&\mathsf{const\ bit}\langle 48\rangle\ \mathsf{bits} := \ldots;\\ &\mathsf{const\ int\ n} := 6;\\ &\mathsf{bit}\langle 8\rangle\ \mathsf{nth\_byte} := \mathsf{bits}[n*8-1{:}(n-1)*8]\end{aligned}$$

The bitstring slice operator $exp[exp_{hi}{:}exp_{lo}]$ computes a slice of the bits of $exp$ from the high bit at $exp_{hi}$ down to the low bit $exp_{lo}$ (inclusive). Since the slice endpoints appear in the type $(\mathsf{bit}\langle exp_{hi} - exp_{lo} + 1\rangle)$ they must be known at compile-time.

Unary operations $\ominus$ and binary operations $\oplus$ are drawn from a set of symbols including standard arithmetic and bitwise operations as well as comparisons and equality. Casts are permitted between numeric types and from record types to header types.

*3.1.4 Statements (fig. 3).* Core P4's statement language is small and mostly standard.

Constants are available for use at the type level, so their initializers must themselves be known at compile-time.

Exit statements abort an entire computation. For example, if we pass an invalid IP header to $g$ in the following program, the exit statement in $f$ causes the second call to never happen:

$$\begin{aligned}&\mathsf{function}\ \{\}\ f(\mathsf{in\ h : hdr\_t})\ \{\mathsf{if}\ (!\mathsf{isValid(h.ip)})\ \{\mathsf{exit}\}\ \mathsf{else}\ \{\ldots\}\}\\ &\mathsf{function}\ \{\}\ g(\mathsf{in\ h : hdr\_t})\ \{f(h); f(h)\}\end{aligned}$$

The return type {} is an empty record type used to represent P4's `void` type.

An instantiation takes the form $X(\overline{exp})\ x$ and creates an object named $x$ by invoking the constructor for the type $X$. In full P4 there are restrictions on what kinds of objects can be instantiated where, but we do not reproduce these rules in Core P4.

*3.1.5 Declarations and programs (fig. 4).* Declarations are partitioned into variable declarations, object declarations, and type declarations. Variable declarations are part of statements, which have already been introduced, and type declarations are essentially types, which are discussed above. This leaves object declarations: tables, controls, and functions.

P4 tables can be thought of as generalizing routing tables and switch statements. Like routing tables on specialized network hardware, they store a list of pattern-matching rules that can be edited at run time. Like switch statements, they may run different code depending on the value of an expression.

| | | | |
|---|---|---|---|
| *decl* | ::= | *var_decl* | variables |
| | \| | *obj_decl* | objects |
| | \| | *typ_decl* | types |
| *var_decl* | ::= | const $\tau\ x := exp$ | constants |
| | \| | $\tau\ x := exp$ | local variables (initialized) |
| | \| | $\tau\ x$ | local variables (uninitialized) |
| | \| | $X(\overline{exp})\ x$ | instantiations |
| *typ_decl* | ::= | typedef $\tau\ X$ | typedefs |
| | \| | enum $X\ \{\overline{f}\}$ | enums |
| | \| | error $\{\overline{f}\}$ | errors |
| | \| | match_kind $\{\overline{f}\}$ | match kinds |
| *obj_decl* | ::= | table $x\ \{\overline{key}\ \overline{act}\}$ | tables |
| | \| | ctrl $X(\overline{d\ x : \tau})(\overline{x : \tau})\ \{\overline{decl}\ stmt\}$ | controls |
| | \| | function $\tau\ x\langle\overline{X}\rangle(\overline{d\ x : \tau})\ \{stmt\}$ | functions |
| *key* | ::= | $exp : x$ | table keys |
| *act* | ::= | $x(\overline{exp}, \overline{x : \tau})$ | actions |
| *prog* | ::= | $\overline{decl}$ | programs |

Fig. 4. Core P4 declarations and programs.

In the following example, a table inspects the packet's destination Ethernet address and either sets its egress (output) port or drops the packet.

```
function {} set_port(in port : bit⟨9⟩) {meta.egress_port = port;}
function {} drop() {meta.drop = true;}
table forward {{hdr.eth.dstAddr : exact}  {set_port(); drop();}}
```

The meta struct contains metadata about the packet, while hdr holds the parsed contents of the packet. The exact annotation on the key indicates that patterns in rules should be matched exactly, as opposed to ranges, longest prefixes, or any other match_kind supported by the architecture.

We do not model table rules in Core P4 and instead overapproximate them by assuming a "control plane" $C$ that deterministically selects an action given an identifier for a table and values for its keys. The identifier is an internal location rather than a table name so that distinct table instantiations arising from the same declaration can have separate rules.

Control declarations include a list of parameters (with directions) and a list of constructor parameters (without directions). The body of the declaration includes a list of declarations followed by a statement, which is typically a block containing several statements. While Core P4 does not impose this restriction, the full P4 language requires tables and other stateful objects to be declared within controls rather than at the top level.

Functions are standard, although recursion is not permitted.

*3.1.6 L-values (fig. 3).* An l-value is an expression that can appear on the left-hand side of an assignment statement. They are built up from variables, array indexing, field lookup, and bitslices. A syntactic distinction between expressions and l-values is not enough in general because of function calls, which require arguments for out or inout parameters to be l-values but make no such imposition on their in arguments. To address this the type system checks whether expressions are assignable (see section 3.2).

*3.1.7 Values and signals (fig. 5).* Record values are standard. A header value augments a record with a validity tag, marking whether the header has been initialized. When parsing a packet into a header, a valid tag is added if it does not already exist. Native functions are available to check for,

| | | | |
|---|---|---|---|
| $val$ | ::= | $b$ | booleans |
| | \| | $n_w$ | integers |
| | \| | $\{\overline{f = val}\}$ | records |
| | \| | header {valid, $\overline{f : \tau = val}$} | headers |
| | \| | $X.f$ | type members |
| | \| | stack $\tau$ $\{\overline{val}\}$ | header stacks |
| | \| | clos$(\epsilon, \overline{X}, \overline{d\ x : \tau}, \tau, \overline{decl}\ stmt)$ | closures |
| | \| | native$(x, \overline{d\ x : \tau}, \tau)$ | built-in functions |
| | \| | table $\ell(\epsilon, \overline{key}, \overline{act})$ | table values |
| | \| | cclos$(\epsilon,$ ctrl$(\overline{d\ x : \tau})(\overline{x_c : \tau_c})$ $\{\overline{decl}\ stmt\})$ | constructor closures |
| $sig$ | ::= | cont | continue normally |
| | \| | return $val$ | return value |
| | \| | exit | exit/reject all enclosing calls |

Fig. 5. Core P4 values and signals. A value, naturally, is the result of evaluating an expression. A signal is the result of evaluating a statement or declaration.

| | | | |
|---|---|---|---|
| $\Gamma$ | ::= | $\Gamma,\ x_1 : \tau_1$ | typing context |
| | \| | $\Gamma,\ X_1 : \tau_1$ | constructor type |
| | \| | [] | |
| $\Delta$ | ::= | $\Delta,\ X_1$ var | type variable and definition context |
| | \| | $\Delta,\ X_1 = \tau_1$ | type definition |
| | \| | [] | |
| $\Sigma$ | : | Var $\rightarrow$ Value | constant context |
| $\sigma$ | : | Loc $\rightarrow$ Value | store |
| $\epsilon$ | : | Var $\rightarrow$ Loc | environment |
| $\Xi$ | : | Loc $\rightarrow$ Type | store typing context |
| $C$ | : | Loc $\times$ Value $\times$ $\overline{\text{PartialActRef}} \rightarrow$ ActRef | control plane |

Fig. 6. Evaluation and typechecking contexts and environments. All function spaces in this figure are restricted to finite partial maps. Stores associate values with locations. Evaluation environments associate locations with variables. A PartialActRef is a function call expression with missing parameters, while an ActRef is an ordinary function call expression.

remove, or add a tag. Header and stack values include their field and element types to facilitate our treatment of undefined reads (see section 3.3.2).

A single closure construct is used to represent function closures and constructed controls, so a closure can contain declarations. A closure includes an environment, but not a store, so that closure calls see updates to mutable variables that were in scope when the closure was created.

Native functions are provided by the architecture in the initial program environment. They always include common operations for manipulating header validity bits and the like, but may also include architecture-specific functionality, for example, hash functions.

$$\text{bit}\langle 16\rangle\ \text{hash\_crc16}\langle \text{T}\rangle(\text{in data} : \text{T});$$

Table closures include an environment for evaluating key expressions and a list of actions. They include a location $\ell$ used as an identifier for the control plane to disambiguate between different instances of the same table declaration.

Signals are used to encode normal and exceptional control-flow: continuing normally, returning a value, or exiting.

| | | | |
|---|---|---|---|
| $\Sigma, \Gamma, \Delta \vdash exp : \tau$ goes $d$ | Expression typing | $\Xi, \Delta \vdash \epsilon : \Gamma$ | Environment typing |
| $\Sigma, \Gamma, \Delta \vdash stmt \dashv \Sigma', \Gamma'$ | Statement typing | $\Xi, \Sigma, \Delta \vdash val : \tau$ | Value typing |
| $\Sigma, \Gamma, \Delta \vdash decl \dashv \Sigma', \Gamma', \Delta'$ | Declaration typing | $\Delta \vdash \rho \preceq \rho'$ | Legal casts |
| $\Xi, \Sigma, \Delta \vdash \sigma$ | Store typing | $\Sigma, \Delta \vdash \tau \rightsquigarrow \tau'$ | Type simplification |
| $\langle \Sigma, exp \rangle \rightsquigarrow v$ | Compile-time evaluation | | |

Fig. 7. Selected judgment signatures from the static semantics.

*3.1.8 Typing and evaluation contexts (fig. 6).* There are four kinds of context used in typechecking Core P4 programs: typing contexts $\Gamma$, type definition contexts $\Delta$, store typing contexts $\Xi$, and constant contexts $\Sigma$. Typing contexts are lists of bindings, giving types to variable names and type names $X$. In particular, if $X$ is a type with a constructor, the type of the constructor will be recorded in $\Gamma$ under the name $X$. Type definition contexts include freely mixed definitions $X = \tau$ and variable markers $X$ var. Store typings are finite partial maps from locations to types. Constant contexts are finite partial maps from variable names to compile-time values.

## 3.2 Static semantics

The static semantics for Core P4 takes care of copy-in copy-out typechecking, compile-time computation in types, generics, type definitions, casts, open enumerations, and extern (native) functions. Surface concerns like type argument inference and implicit cast insertion are handled in the Petr4 interpreter but omitted here (see Section 4 for details).

Typing judgments are given in fig. 7. The first three judgments are the top-level program typing judgments. Store, environment, and value typing are not used to typecheck programs but are necessary in order to formulate our type safety theorem. The type simplification judgment replaces type variables in $\tau$ with their definitions in $\Delta$ and performs compile-time evaluation on any expressions that appear in $\tau$. The compile-time evaluation judgment only needs a constant environment and an expression.

The expression typing judgment produces a direction indicating whether the expression is assignable (goes inout) or not (goes in.) Sometimes we need the type of an expression but do not care about its direction. In such a situation the expression typing judgment may be written $\Sigma, \Gamma, \Delta \vdash exp : \tau$, leaving off the direction annotation goes $d$.

Statement typechecking produces a new constant context and a new typing context. Declaration typechecking produces new constant and typing contexts, as for statements, but it also produces an updated type variable context to hold any new type definitions.

Our type soundness proof assumes that function and control bodies always return a value. In the implementation, a simple static analysis integrated into statement typechecking ensures that this is the case. We omit it here in order to avoid cluttering up the typing rules.

The type simplification judgment replaces type variables with their definitions in $\Delta$ and evaluates expressions occurring in types. Here is an example of it substituting a definition for the type variable $C$, including recursive substitutions for the type variable $B$ and the expression $c + 1$.

$$c := 7,\ C = \mathsf{bit}\langle c + 1 \rangle,\ B = C \ \vdash\ B \rightsquigarrow \mathsf{bit}\langle 8 \rangle$$

*3.2.1 Expression typing (fig. 7).* The expression typing judgment is defined in fig. 8. It is designed to only ever output types in a canonical form with no unevaluated expressions and no free variables except the ones declared with $X$ var in $\Delta$.

The typing rules for l-values (arrays, bitslices, fields) check the direction $d$ of their "root" subexpression. The only rule that produces $d \neq$ in is T-Var, which requires $x$ to not be in the constant context. The types of unary and binary operators are determined by a type interpretation function

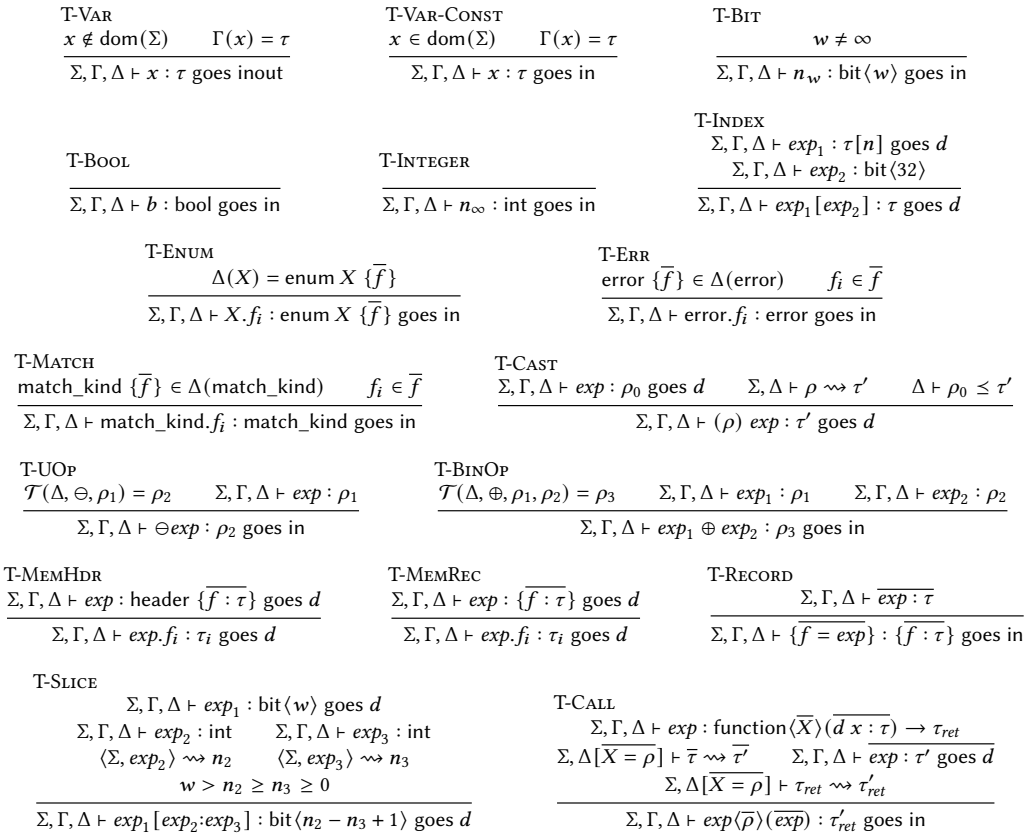



Fig. 8. Expression typing rules.

$\mathcal{T}$. Array indexes are not required to be compile time known and are not bounds checked. By contrast, the endpoints of a bit slice receive both treatments because the type of a slice depends on the values of its endpoints and types should only depend on compile-time values. Bounds checking is a bonus, since the endpoints are already evaluated. The function call rule uses type simplification to substitute type arguments into parameter types and return types.

*3.2.2 Statement typing (fig. 9).* The typing rules for statements, defined in fig. 9, are largely standard. The relation $\Sigma, \Gamma, \Delta \vdash stmt \dashv \Sigma, \Gamma$ holds when a statement executed in the contexts on the left side will produce a final state satisfying the contexts on the right side. The constant context $\Sigma$ appears on the right because constants can be declared in statements, while $\Gamma$ appears because variables can be declared in statements.

The assignment rule TS-Assign checks that the expression on the left side has direction inout, which means (as we saw in the expression typing rules) that it is an l-value. The return rule TS-Ret checks that the type of the value being returned agrees with the type of the special identifier return. Declaration typing rules (see fig. 11) insert a type for return before typechecking the bodies of functions and controls.

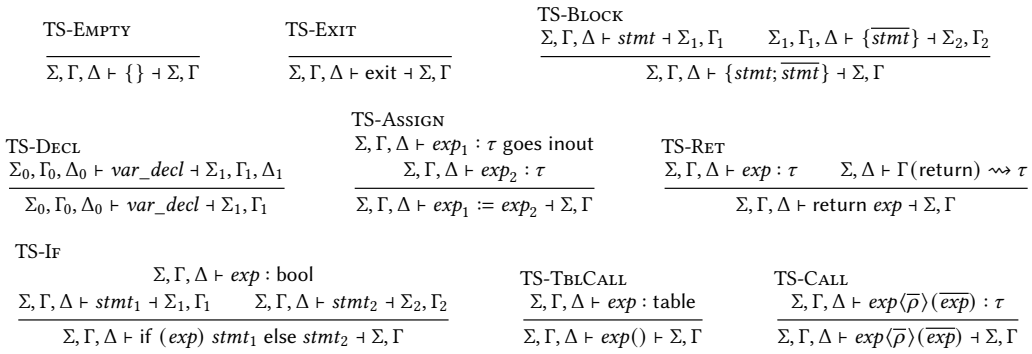



Fig. 9. Statement typing rules.

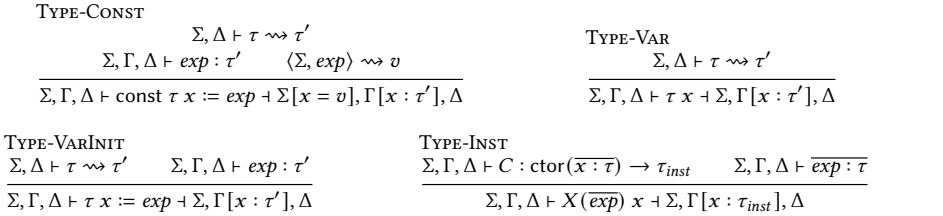



Fig. 10. Variable declaration typing rules.

*3.2.3 Variable declaration rules (fig. 10).* Variable declarations introduce new variables and can be used as statements. Their typing relation includes an output type context for uniformity with other declarations but they do not bind new types.

*3.2.4 Object declaration rules (fig. 11).* Table typechecking checks keys and match kinds. An action *act* is a partial application of a function, so the auxiliary judgment act_ok checks the action like a function call but allows any number of arguments to be left off. Omitted arguments are the responsibility of the control plane.

The typing rules for controls and functions use a special return identifier to check return statements within the body of the declaration. As discussed earlier, the type {} is an empty record type representing P4's `void` return type.

### 3.3 Dynamic semantics

The dynamic semantics for Core P4 is defined in a big-step style. Figure 12 gives the types of the main judgments. Local state is split into a store and an environment to implement the scoping of mutable variables. The environment maps names of variables to store locations, and the store maps locations to values. This decoupling allows closures to witness updates to mutable variables saved in their environments.

Morally speaking, P4 programs are deterministic. The semantics of Core P4 introduces nondeterminism in a few places to simplify the presentation or to model architecture-dependent behavior. For example, the result of reading an invalid header is an undefined value, which may vary from target to target and even from read to read within a program. We write $\text{havoc}(\tau)$ to indicate an operation producing an arbitrary value of type $\tau$. Match-action evaluation uses the control plane $C$

$$\textsc{T-TableDecl}\quad \frac{\Sigma, \Gamma, \Delta \vdash \overline{exp_k : \tau_k} \qquad \Sigma, \Gamma, \Delta \vdash \overline{x_k : \text{match\_kind}} \qquad \Sigma, \Gamma, \Delta \vdash \overline{act\ \text{act\_ok}}}{\Sigma, \Gamma, \Delta \vdash \text{table}\ x\ \{\overline{exp_k : x_k}\ \overline{act}\} \dashv \Sigma, \Gamma[x : \text{table}], \Delta}$$

$$\textsc{T-CtrlDecl}\quad \frac{\Sigma, \Delta \vdash \overline{\tau} \rightsquigarrow \overline{\tau'} \qquad \Sigma, \Gamma[\overline{x_c : \tau_c}][\overline{x : \tau}], \Delta \vdash \overline{decl} \dashv \Sigma_1, \Gamma_1, \Delta_1 \qquad \Sigma_1, \Gamma_1[\text{return} : \{\}], \Delta_1 \vdash stmt \dashv \Sigma_2, \Gamma_2}{\Sigma, \Gamma, \Delta \vdash \text{ctrl}\ X(\overline{d\ x : \tau'})(\overline{x_c : \tau'_c})\ \{\overline{decl}\ stmt\} \dashv \Sigma, \Gamma[X : \text{ctor}(\overline{x_c : \tau'_c}) \to \text{function}(\overline{d\ x : \tau'}) \to \{\}], \Delta}$$

$$\textsc{T-FuncDecl}\quad \frac{\begin{array}{c}\Gamma_1 = \Gamma[\overline{x_i : \tau'_i}, \text{return} : \tau'] \qquad \Delta_1 = \Delta[\overline{X\ \text{var}}] \qquad \Sigma, \Delta_1 \vdash \overline{\tau_i} \rightsquigarrow \overline{\tau'_i} \\ \Sigma, \Delta_1 \vdash \tau \rightsquigarrow \tau' \qquad \Sigma, \Gamma_1, \Delta_1 \vdash stmt \dashv \Sigma_2, \Gamma_2\end{array}}{\Sigma, \Gamma, \Delta \vdash \text{function}\ \tau\ x\langle\overline{X}\rangle(\overline{d\ x_i : \tau_i})\ \{stmt\} \dashv \Sigma, \Gamma[x : \text{function}\langle\overline{X}\rangle(\overline{d\ x_i : \tau'_i}) \to \tau'], \Delta}$$

Fig. 11. Object declaration typing rules.

| | |
|---|---|
| $\langle \Delta, \sigma, \epsilon, \tau \rangle \Downarrow_\tau \tau'$ | Type simplification |
| $\langle C, \Delta, \sigma, \epsilon, \overline{d\ x : \tau := exp} \rangle \Downarrow_{copy} \langle \sigma', \overline{x \mapsto \ell}, \overline{lval := \ell} \rangle$ | Copy-in copy-out |
| $\langle C, \Delta, \sigma, \epsilon, lval := val \rangle \Downarrow_{write} \sigma'$ | L-value assignment |
| $\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow_{lval} \langle \sigma', lval \rangle$ | L-value evaluation |
| $\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', val \rangle$ | Expression evaluation |
| $\langle C, x, \overline{val : x} \rangle \Downarrow_{match} x(\overline{exp})$ | Match-action evaluation |
| $\langle C, \Delta, \sigma, \epsilon, stmt \rangle \Downarrow \langle \sigma', \epsilon', sig \rangle$ | Statement evaluation |
| $\langle C, \Delta, \sigma, \epsilon, decl \rangle \Downarrow \langle \Delta', \sigma', \epsilon', sig \rangle$ | Declaration evaluation |

Fig. 12. Selected judgment signatures from the dynamic semantics. Abusing notation, we let expression evaluation judgment output a *sig* instead of a *val*, and likewise for L-value evaluation.

to select from the table's actions (rather than defining an algorithm for selecting it from a list of forwarding rules). We give tables unique identifiers for control plane use by reusing locations $\ell$, which are also generated non-deterministically, although this is not essential.

Statements evaluate to signals, which indicate how control flow should proceed. Expressions evaluate to signals as well but with values *val* in place of the cont signal. The signals are how Core P4 models non-standard control flow. To save space, we elide the "unwinding" rules for handling signals other than cont or *val* in most places. For each intermediate computation with outputs $\sigma$ and $\epsilon$ if that computation terminates in exit or return *val*, the overall computation freezes the state at $\langle \sigma, \epsilon \rangle$ and propagates the signal.

*3.3.1 Copy-in copy-out rules (fig. 13).* P4 uses a copy-in copy-out convention for function calls. This convention guarantees that distinct variable names within a function refer to distinct storage locations. This means P4 compilers and static analyses never have to account for aliasing. The following example shows how copy-in copy-out handles aliasing of function arguments. The {} before $f$ is its return type, a record with no fields (i.e., unit).

$$\begin{array}{l}\text{function}\ \{\}\ f(\text{inout bit}\langle 8\rangle\ src, \text{inout bit}\langle 8\rangle\ dst)\ \{dst := src + 1;\ src := 0\} \\ x := 1; \\ f(x, x);\end{array}$$

$$\textsc{CopyIn}\quad \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma', val\rangle \qquad \ell \text{ fresh}}{\langle C, \Delta, \sigma, \epsilon, \text{in } x : \tau := exp\rangle \Downarrow_{copy} \langle \sigma'[\ell \mapsto val], x \mapsto \ell, [\,]\rangle}$$

$$\textsc{CopyOut}\quad \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow_{lval} \langle \sigma', lval\rangle \qquad \ell \text{ fresh}}{\langle C, \Delta, \sigma, \epsilon, \text{out } x : \tau := exp\rangle \Downarrow_{copy} \langle \sigma[\ell \mapsto \text{init}_\Delta\ \tau], x \mapsto \ell, [lval := \ell]\rangle}$$

$$\textsc{CopyInOut}\quad \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow_{lval} \langle \sigma_1, lval\rangle \qquad \langle \Delta, \sigma_1, \epsilon, lval\rangle \Downarrow \langle \sigma_2, val\rangle \qquad \ell \text{ fresh}}{\langle C, \Delta, \sigma, \epsilon, \text{inout } x : \tau := exp\rangle \Downarrow_{copy} \langle \sigma_2[\ell \mapsto val], x \mapsto \ell, [lval := \ell]\rangle}$$

Fig. 13. Copy-in and copy-out operations. We define them for single arguments and they are lifted to lists of arguments in the obvious way.

In a call-by-reference language $x$ would be 0 after the call to $f$. In a call-by-value language, it would still be 1. In P4, however, $x$ will be 2. A function call creates temporaries for storing its arguments for each call and copies the temporaries back, in order, after the body of the function finishes. In the example *dst* comes last in the parameter list of $f$, so $x$ ends up with the *dst* value (2) overwriting the *src* value (0).

*3.3.2 Expression evaluation (fig. 14).* Unary operations, binary operations, and casts are axiomatized. Rather than spell out all the legal casts or arithmetic expressions, we assume we have typing and evaluation oracles for each of them which agree. For unary and binary operations, this means that there is a typing function $\mathcal{T}$ and an evaluation function $\mathcal{E}$. For casts, this means there is agreement between a casting check $\Delta \vdash \tau \preceq \tau'$ and a casting function $\text{cast}(\Sigma, val, \tau)$.

The P4 specification allows programs to produce "undefined values" in certain situations. This is substantially more restrictive than the concept of "undefined behavior" in C, which has notoriously confusing semantics [Wang et al. 2012]. Our E-HdrMemUnref rule introduces an undefined (havoc'd) value when a program attempts to read from an invalid header, but does not affect any other program state.

The full P4 expression language includes built-in functions for operations such as accessing header validity bits. Core P4 models these functions using native functions, which we assume are already in the context at the start of program execution and which are evaluated by appealing to an interpretation $\mathcal{N}$.

*3.3.3 Variable declaration evaluation (fig. 15).* The next collection of formal rules handles variable declarations. Constants and regular values are not distinguished at run time. The most interesting rule is E-Inst for instantiations. It produces a closure without executing any additional code, saving the constructor arguments in the store and closure environment.

*3.3.4 Object declaration evaluation (fig. 16).* The object declarations create closures from declarations of tables, controls, and functions. All closures save a copy of the environment, but do not save a copy of the store. Control and function closures are standard. Table closures save the fresh location of the table for use by the control plane in disambiguating multiple tables instantiated from a single declaration. Table closures also save the table key expressions and the list of actions available to the table for use in matching.

*3.3.5 Statement evaluation (fig. 17).* The rules for statement evaluation are mostly standard. The most interesting rule is E-Call-Table, which handles table invocation. It first evaluates the key and then uses the control-plane to locate a matching action, and executes the body of the action to

E-Int
$$\frac{}{\langle C, \Delta, \sigma, \epsilon, n_w \rangle \Downarrow \langle \sigma, n_w \rangle}$$

E-Bool
$$\frac{}{\langle C, \Delta, \sigma, \epsilon, b \rangle \Downarrow \langle \sigma, b \rangle}$$

E-TypMem
$$\frac{}{\langle C, \Delta, \sigma, \epsilon, X.f \rangle \Downarrow \langle \sigma, X.f \rangle}$$

E-Var
$$\frac{\epsilon(x) = \ell \qquad \sigma(\ell) = val}{\langle C, \Delta, \sigma, \epsilon, x \rangle \Downarrow \langle \sigma, val \rangle}$$

E-Cast
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', val \rangle \qquad \langle \Delta, \sigma, \epsilon, \tau \rangle \Downarrow_\tau \tau'}{\langle C, \Delta, \sigma, \epsilon, (\tau) exp \rangle \Downarrow \langle \sigma', \mathsf{cast}(\Delta, val, \tau') \rangle}$$

E-Uop
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', val \rangle}{\langle C, \Delta, \sigma, \epsilon, \ominus exp \rangle \Downarrow \langle \sigma', \mathcal{E}(\ominus, val) \rangle}$$

E-BinOp
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp_1 \rangle \Downarrow \langle \sigma_1, val_1 \rangle \qquad \langle C, \Delta, \sigma_1, \epsilon, exp_2 \rangle \Downarrow \langle \sigma_2, val_2 \rangle}{\langle C, \Delta, \sigma, \epsilon, exp_1 \oplus exp_2 \rangle \Downarrow \langle \sigma_2, \mathcal{E}(\oplus, val_1, val_2) \rangle}$$

E-Rec
$$\frac{\langle C, \Delta, \sigma, \epsilon, \overline{exp} \rangle \Downarrow \langle \sigma', \overline{val} \rangle}{\langle C, \Delta, \sigma, \epsilon, \{\overline{f = exp}\} \rangle \Downarrow \langle \sigma', \{\overline{f = val}\} \rangle}$$

E-RecMem
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', \{\overline{f : \tau = val}\} \rangle}{\langle C, \Delta, \sigma, \epsilon, exp.f_i \rangle \Downarrow \langle \sigma', val_i \rangle}$$

E-Index
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp_1 \rangle \Downarrow \langle \sigma_1, \mathsf{stack}\ \tau\ \{\overline{val}\} \rangle \qquad \langle C, \Delta, \sigma_1, \epsilon, exp_2 \rangle \Downarrow \langle \sigma_2, n_{32} \rangle \qquad 0 \le n < \mathsf{len}(\overline{val})}{\langle C, \Delta, \sigma, \epsilon, exp_1[exp_2] \rangle \Downarrow \langle \sigma_2, val_n \rangle}$$

E-IndexOOB
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp_1 \rangle \Downarrow \langle \sigma_1, \mathsf{stack}\ \tau\ \{\overline{val}\} \rangle \qquad \langle C, \Delta, \sigma_1, \epsilon, exp_2 \rangle \Downarrow \langle \sigma_2, n_{32} \rangle \qquad n \ge \mathsf{len}(\overline{val})}{\langle C, \Delta, \sigma, \epsilon, exp_1[exp_2] \rangle \Downarrow \langle \sigma_2, \mathsf{havoc}(\tau) \rangle}$$

E-HdrMem
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', \mathsf{header}\ \{\mathsf{valid},\ \overline{f : \tau = val}\} \rangle}{\langle C, \Delta, \sigma, \epsilon, exp.f_i \rangle \Downarrow \langle \sigma', val_i \rangle}$$

E-HdrMemUndef
$$\frac{\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', \mathsf{header}\ \{!\mathsf{valid},\ \overline{f : \tau = val}\} \rangle}{\langle C, \Delta, \sigma, \epsilon, exp.f_i \rangle \Downarrow \langle \sigma', \mathsf{havoc}(\tau_i) \rangle}$$

E-Call-DeclExit
$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, \mathsf{clos}(\epsilon_c, \overline{X}, \overline{d\ x : \tau}, \tau, \overline{decl}\ stmt) \rangle \\ \langle \Delta[\overline{X = \rho}], \sigma, \epsilon, \overline{\tau} \rangle \Downarrow_\tau \overline{\tau'} \\ \langle C, \Delta, \sigma_1, \epsilon, \overline{d\ x : \tau' := exp} \rangle \Downarrow_{copy} \langle \sigma_2, \overline{x \mapsto \ell}, \overline{lval := \ell} \rangle \\ \langle C, \Delta[\overline{X = \rho}], \sigma_2, \epsilon_c[\overline{x \mapsto \ell}], \overline{decl} \rangle \Downarrow \langle \Delta_2, \sigma_3, \epsilon_2, \mathsf{exit} \rangle \\ \langle C, \Delta, \sigma_3, \epsilon, \overline{lval := \sigma_3(\ell)} \rangle \Downarrow_{write} \sigma_4\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp\langle\overline{\rho}\rangle(\overline{exp}) \rangle \Downarrow \langle \sigma_4, \mathsf{exit} \rangle}$$

E-Call-StmtExit
$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, \mathsf{clos}(\epsilon_c, \overline{X}, \overline{d\ x : \tau}, \tau, \overline{decl}\ stmt) \rangle \\ \langle \Delta[\overline{X = \rho}], \sigma, \epsilon, \overline{\tau} \rangle \Downarrow_\tau \overline{\tau'} \\ \langle C, \Delta, \sigma_1, \epsilon, \overline{d\ x : \tau := exp} \rangle \Downarrow_{copy} \langle \sigma_2, \overline{x \mapsto \ell}, \overline{lval := \ell} \rangle \\ \langle C, \Delta[\overline{X = \rho}], \sigma_2, \epsilon_c[\overline{x \mapsto \ell}], \overline{decl} \rangle \Downarrow \langle \Delta_2, \sigma_3, \epsilon_2, \mathsf{cont} \rangle \\ \langle C, \Delta_2, \sigma_3, \epsilon_2, stmt \rangle \Downarrow \langle \sigma_4, \epsilon_3, \mathsf{exit} \rangle \\ \langle C, \Delta, \sigma_4, \epsilon, \overline{lval := \sigma_4(\ell)} \rangle \Downarrow_{write} \sigma_5\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp\langle\overline{\rho}\rangle(\overline{exp}) \rangle \Downarrow \langle \sigma_5, \mathsf{exit} \rangle}$$

E-CallN
$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, \mathsf{native}(x, \overline{d\ x : \tau}, \tau) \rangle \\ \langle C, \Delta, \sigma_1, \epsilon, \overline{d\ x : \tau := exp} \rangle \Downarrow_{copy} \langle \sigma_2, \overline{x \mapsto \ell}, \overline{lval := \ell} \rangle \\ \mathcal{N}(x, \sigma_2, [\overline{x \mapsto \ell}]) = \langle \sigma_3, val \rangle \\ \langle C, \Delta, \sigma_3, \epsilon, \overline{lval := \sigma_3(\ell)} \rangle \Downarrow_{write} \sigma_4\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp(\overline{exp}) \rangle \Downarrow \langle \sigma_4, val \rangle}$$

E-Call
$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, \mathsf{clos}(\epsilon_c, \overline{X}, \overline{d\ x : \tau}, \tau, \overline{decl}\ stmt) \rangle \\ \langle \Delta[\overline{X = \rho}], \sigma, \epsilon, \overline{\tau} \rangle \Downarrow_\tau \overline{\tau'} \\ \langle C, \Delta, \sigma_1, \epsilon, \overline{d\ x : \tau' := exp} \rangle \Downarrow_{copy} \langle \sigma_2, \overline{x \mapsto \ell}, \overline{lval := \ell} \rangle \\ \langle C, \Delta[\overline{X = \rho}], \sigma_2, \epsilon_c[\overline{x \mapsto \ell}], \overline{decl} \rangle \Downarrow \langle \Delta_2, \sigma_3, \epsilon_2, \mathsf{cont} \rangle \\ \langle C, \Delta_2, \sigma_3, \epsilon_2, stmt \rangle \Downarrow \langle \sigma_4, \epsilon_3, \mathsf{return}\ val \rangle \\ \langle C, \Delta, \sigma_4, \epsilon, \overline{lval := \sigma_4(\ell)} \rangle \Downarrow_{write} \sigma_5\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp\langle\overline{\rho}\rangle(\overline{exp}) \rangle \Downarrow \langle \sigma_5, val \rangle}$$

E-Slice
$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp_1 \rangle \Downarrow \langle \sigma_1, n_w \rangle \\ \langle C, \Delta, \sigma_1, \epsilon, exp_2 \rangle \Downarrow \langle \sigma_2, p_\infty \rangle \\ \langle C, \Delta, \sigma_2, \epsilon, exp_3 \rangle \Downarrow \langle \sigma_3, q_\infty \rangle\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp_1[exp_2{:}exp_3] \rangle \Downarrow \langle \sigma_3, n_w[p{:}q] \rangle}$$

Fig. 14. Semantics for expressions.

$$\textsc{E-Const}\ \frac{\langle C, \Delta, \sigma, \epsilon, \tau\ x := exp\rangle \Downarrow \langle \Delta, \sigma_1, \epsilon_1, \text{cont}\rangle}{\langle C, \Delta, \sigma, \epsilon, \text{const}\ \tau\ x := exp\rangle \Downarrow \langle \Delta, \sigma_1, \epsilon_1, \text{cont}\rangle}$$

$$\textsc{E-VarDecl}\ \frac{\ell \text{ fresh} \qquad \langle \Delta, \sigma, \epsilon, \tau\rangle \Downarrow \tau'}{\langle C, \Delta, \sigma, \epsilon, \tau\ x\rangle \Downarrow \langle \Delta, \sigma[\ell := \text{init}_\Delta\ \tau'], \epsilon[x \mapsto \ell], \text{cont}\rangle}$$

$$\textsc{E-VarInit}\ \frac{\ell \text{ fresh} \qquad \langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, val\rangle}{\langle C, \Delta, \sigma, \epsilon, \tau\ x := exp\rangle \Downarrow \langle \Delta, \sigma_1[\ell := val], \epsilon[x \mapsto \ell], \text{cont}\rangle}$$

$$\textsc{E-Inst}\ \frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, X\rangle \Downarrow \langle \sigma_1, \text{cclos}(\epsilon_{cc}, \text{ctrl}(\overline{d\ x : \tau})(\overline{x_c : \tau_c})\ \{\overline{decl}\ stmt\})\rangle \\ \langle C, \Delta, \sigma_1, \epsilon, \overline{exp}\rangle \Downarrow \langle \sigma_2, \overline{val_c}\rangle \\ \overline{\ell_c}, \ell \text{ fresh} \qquad val = \text{clos}(\epsilon_{cc}[\overline{x_c \mapsto \ell_c}], \langle\rangle, \overline{d\ x : \tau}, \{\}, \overline{decl}\ stmt)\end{array}}{\langle C, \Delta, \sigma, \epsilon, X(\overline{exp})\ x\rangle \Downarrow \langle \Delta, \sigma_2[\overline{\ell_c \mapsto val_c}][\ell \mapsto val], \epsilon[x \mapsto \ell], \text{cont}\rangle}$$

Fig. 15. Semantics for variable declarations.

$$\textsc{E-TableDecl}\ \frac{\ell \text{ fresh} \qquad val = \text{table}\ \ell(\epsilon, \overline{key}, \overline{act})}{\langle C, \Delta, \sigma, \epsilon, \text{table}\ x\ \{\overline{key}\ \overline{act}\}\rangle \Downarrow \langle \Delta, \sigma[\ell \mapsto val], \epsilon[x \mapsto \ell], \text{cont}\rangle}$$

$$\textsc{E-CtrlDecl}\ \frac{\ell \text{ fresh} \qquad \langle \Delta, \sigma, \epsilon, \overline{\tau_c}\rangle \Downarrow_\tau \overline{\tau'_c} \qquad \langle \Delta, \sigma, \epsilon, \overline{\tau}\rangle \Downarrow_\tau \overline{\tau'} \qquad val = \text{cclos}(\epsilon, \text{ctrl}(\overline{d\ x : \tau'})(\overline{x_c : \tau'_c})\ \{\overline{decl}\ stmt\})}{\langle C, \Delta, \sigma, \epsilon, \text{ctrl}\ X(\overline{d\ x : \tau})(\overline{x_c : \tau_c})\ \{\overline{decl}\ stmt\}\rangle \Downarrow \langle \Delta, \sigma[\ell \mapsto val], \epsilon[X \mapsto \ell], \text{cont}\rangle}$$

$$\textsc{E-FuncDecl}\ \frac{\ell \text{ fresh} \qquad \langle \Delta[\overline{X \text{ var}}], \sigma, \epsilon, \overline{\tau_i}\rangle \Downarrow_\tau \overline{\tau'_i} \qquad \langle \Delta[\overline{X \text{ var}}], \sigma, \epsilon, \tau\rangle \Downarrow_\tau \tau' \qquad val = \text{clos}(\epsilon, \overline{X}, \overline{d\ x_i : \tau'_i}, \tau', stmt)}{\langle C, \Delta, \sigma, \epsilon, \text{function}\ \tau\ x\langle\overline{X}\rangle(\overline{d\ x_i : \tau_i})\ \{stmt\}\rangle \Downarrow \langle \Delta, \sigma[\ell \mapsto val], \epsilon[x \mapsto \ell], \text{cont}\rangle}$$

Fig. 16. Semantics for object declarations.

obtain the final result. For simplicity, in Core P4, we assume that tables have a default action, so they cannot "miss."

### 3.4 Putting it all together

The static and dynamic semantics presented thus far omit type declarations. Full rules are in the appendix, but type declarations are simple. Aside from the open enum type declarations, which add new members to their type, type declarations just add new type definitions to the type context.

### 3.5 Type soundness and termination

Big-step semantics fail to distinguish between programs that "go wrong" and programs that run forever. For a language with recursion or loops, this can complicate the proof of a useful type soundness result. Fortunately, the parser-free fragment of P4 has neither, so we can prove that all well-typed expressions and statements evaluate to a final value of appropriate type. The main theorem shows this for statements.

Theorem 3.1. *Let* $\langle C, \Delta, \sigma, \epsilon, stmt\rangle$ *be an initial configuration and take contexts* $\Xi, \Sigma, \Sigma', \Gamma, \Gamma', \Delta$. *Suppose*

*(1)* $\Xi, \Sigma, \Delta \vdash \sigma$,
*(2)* $\Xi \vdash \epsilon : \Gamma$, *and*
*(3)* $\Sigma, \Gamma, \Delta \vdash stmt \dashv \Sigma', \Gamma'$.

$$\textsc{E-Empty}\ \frac{}{\langle C, \Delta, \sigma, \epsilon, \{\}\rangle \Downarrow \langle \sigma, \epsilon, \mathsf{cont}\rangle} \qquad \textsc{E-Exit}\ \frac{}{\langle C, \Delta, \sigma, \epsilon, \mathsf{exit}\rangle \Downarrow \langle \sigma, \epsilon, \mathsf{exit}\rangle}$$

$$\textsc{E-IfT}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, \mathsf{true}\rangle \quad \langle C, \Delta, \sigma_1, \epsilon, stmt_1\rangle \Downarrow \langle \sigma_2, \epsilon_2, sig\rangle}{\langle C, \Delta, \sigma, \epsilon, \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2\rangle \Downarrow \langle \sigma_2, \epsilon, sig\rangle}$$

$$\textsc{E-IfF}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, \mathsf{false}\rangle \quad \langle C, \Delta, \sigma_1, \epsilon, stmt_2\rangle \Downarrow \langle \sigma_2, \epsilon_2, sig\rangle}{\langle C, \Delta, \sigma, \epsilon, \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2\rangle \Downarrow \langle \sigma_2, \epsilon, sig\rangle}$$

$$\textsc{E-Block}\ \frac{\langle C, \Delta, \sigma, \epsilon, stmt\rangle \Downarrow \langle \sigma_1, \epsilon_1, \mathsf{cont}\rangle \quad \langle C, \Delta, \sigma_1, \epsilon_1, \{\overline{stmt}\}\rangle \Downarrow \langle \sigma_2, \epsilon_2, sig\rangle}{\langle C, \Delta, \sigma, \epsilon, \{stmt, \overline{stmt}\}\rangle \Downarrow \langle \sigma_2, \epsilon, sig\rangle}$$

$$\textsc{E-Return}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, val\rangle}{\langle C, \Delta, \sigma, \epsilon, \mathsf{return}\ exp\rangle \Downarrow \langle \sigma_1, \epsilon, \mathsf{return}\ val\rangle}$$

$$\textsc{E-Assign}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp_1\rangle \Downarrow_{lval} \langle \sigma_1, lval\rangle \quad \langle C, \Delta, \sigma_1, \epsilon, exp_2\rangle \Downarrow \langle \sigma_2, val\rangle \quad \langle C, \Delta, \sigma_2, \epsilon, lval := val\rangle \Downarrow_{write} \sigma_3}{\langle C, \Delta, \sigma, \epsilon, exp_1 := exp_2\rangle \Downarrow \langle \sigma_3, \epsilon, \mathsf{cont}\rangle}$$

$$\textsc{E-Call-Table}\ \frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, \mathsf{table}\ \ell(\epsilon_c, \overline{exp_{key} : x}, \overline{x_{act}(\overline{exp_s}, \overline{x_c : \tau})})\rangle \\ \langle C, \Delta, \sigma_1, \epsilon_c, \overline{exp_{key}}\rangle \Downarrow \langle \sigma_2, \overline{val_{key}}\rangle \\ \langle C, \ell, \overline{val_{key} : x}, \overline{x_{act}(\overline{x_c : \tau})}\rangle \Downarrow_{match} x_{act}(\overline{exp_c}) \\ \langle C, \Delta, \sigma_2, \epsilon_c, x_{act}(\overline{exp_s}, \overline{exp_c})\rangle \Downarrow \langle \sigma_3, \epsilon_c', \mathsf{cont}\rangle\end{array}}{\langle C, \Delta, \sigma, \epsilon, exp()\rangle \Downarrow \langle \sigma_3, \epsilon, \mathsf{cont}\rangle}$$

$$\textsc{E-VarDecl}\ \frac{\langle C, \Delta, \sigma, \epsilon, var\_decl\rangle \Downarrow \langle \Delta', \sigma', \epsilon', \mathsf{cont}\rangle}{\langle C, \Delta, \sigma, \epsilon, var\_decl\rangle \Downarrow \langle \sigma', \epsilon', \mathsf{cont}\rangle} \qquad \textsc{E-Call}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp\langle\overline{\rho}\rangle(\overline{exp})\rangle \Downarrow \langle \sigma', sig\rangle}{\langle C, \Delta, \sigma, \epsilon, exp\langle\overline{\rho}\rangle(\overline{exp})\rangle \Downarrow \langle \sigma', \epsilon, sig\rangle}$$

Fig. 17. Semantics for statements.

*Then there exists a final configuration* $\langle \sigma', \epsilon', sig\rangle$ *and a store typing* $\Xi' \supseteq \Xi$ *such that*

*(1)* $\langle C, \Delta, \sigma, \epsilon, stmt\rangle \Downarrow \langle \sigma', \epsilon', sig\rangle$,
*(2)* $\Xi', \Sigma', \Gamma', \Delta \vdash \sigma'$,
*(3)* $\Xi', \Sigma', \Gamma', \Delta \vdash \epsilon' : \Gamma'$, *and*
*(4) if* $sig = \mathsf{return}\ val$ *then there is a type* $\tau$ *such that* $\Gamma(\mathsf{return}) = \tau$ *and* $\Xi', \Sigma', \Delta \vdash val : \tau$.

The proof is a simple but tedious proof by logical relations, given in the extended version of the paper. It includes additional supporting definitions and analogous theorems for expressions and variable declarations. Note that this is a "weak termination" result: it states that a final configuration exists, but does not (and cannot, in the language of big-step semantics) say that all possible ways of evaluating a program will terminate.

## 4 IMPLEMENTATION

This section presents Petr4's definitional interpreter. Unlike the mathematical semantics for Core P4 developed in the last section, which only models a subset of the language, our implementation is designed to handle the full $P4_{16}$ language, with a few caveats and limitations discussed below.

*Overview.* Figure 18 (a) depicts the architecture of the interpreter, as well as the way that programs and packets flow through it. We implemented Petr4 in OCaml, using the Menhir parser generator, the Jane Street `Core` library, and the `js_of_ocaml` OCaml-to-Javascript compiler. In total, the Petr4 implementation runs 13KLoC (as reported by `cloc`) of which 1.5KLoC implements lexing and parsing, 1.5KLoC defines syntax, 4KLoC implements typechecking, and 4.5KLoC implements evaluation/interpretation. The remaining 1.5KLoC is miscellaneous utility code.

*Lexer and parser.* The $P4_{16}$ specification defines the syntax of the language with an EBNF grammar. Unfortunately the grammar cannot be parsed by any LALR(1) parser due to a conflict between generics and bit shifts over the symbols '<' and '>' following identifiers. As a workaround, the specification separates the tokens for identifiers into two categories:

> *The grammar is actually ambiguous, so the lexer and the parser must collaborate for parsing the language. In particular, the lexer must be able to distinguish two kinds of identifiers: type names previously introduced (*`TYPE_IDENTIFIER` *tokens) [and] regular identifiers (*`IDENTIFIER` *token).*

Hence, the parser must keep track of rudimentary type information as well as lexical scope, so that the lexer can produce the correct tokens. We follow Jourdan and Pottier's approach for implementing a parser for C11 in Menhir [Jourdan and Pottier 2017]: the parser maintains a simple context to keep track of the set of type names, and we wrap a simple lexer that produces `NAME` tokens with a second lexer that uses the context to rewrite those tokens into `IDENTIFIER` or `TYPE_IDENTIFIER` as appropriate.

*Type checker.* P4 surface syntax leaves much to the imagination. Function calls may omit type arguments which have to be inferred. Expressions may be used at the "wrong" type, omitting implicit casts which have to be inserted by the typechecker. Widths in numeric types, as in Core P4, may be expressions which have to be evaluated. The Petr4 type checker addresses all these issues, converting programs written in an ambiguous surface syntax into an unambiguous internal syntax. In the typed internal syntax tree, all nodes are tagged with their type and all casts and type arguments are made explicit. Compile-time known expressions are replaced with their values. The Core P4 language is closer to this fully elaborated and typed syntax, although it does retain an account of compile-time evaluation.

The $P4_{16}$ specification does not precisely define a type system for the language. Key questions such as how type inference works, where casts may be automatically inserted, and whether type equivalence is nominal or structural are not addressed. As an example, the specification uses the following text to introduce "don't care" types:

> *The "don't care" identifier (_) can only be used for an* `out` *function/method argument, when the value of [sic] returned in that argument is ignored by subsequent computations. When used in generic functions or methods, the compiler may reject the program if it is unable to infer a type for the don't care argument.*

However, aside from a brief mention of the Hindley-Milner inference algorithm [Damas 1984], there is no explanation of when the compiler should, if ever, be able to infer a missing type argument. In practice, p4c does use a full Hindley-Milner implementation to infer type arguments and check type equality, which has been the source of surprising typechecking bugs [Foster 2019]. What is more surprising is that Hindley-Milner is unnecessary for $P4_{16}$. Without solid metatheory available, the language specification restricts type abstraction to only a few language constructs. In this simple setting, we found that a much simpler inference algorithm can get the job done.

The Petr4 inference algorithm is inspired by local type inference [Pierce and Turner 2000], but even LTI is a little heavyweight for the present state of P4 generics. Where LTI collects type-type constraints of the form $\tau_1 = \tau_2$, Petr4 is able to stick to variable-type constraints of the form $X = \tau$. At a call site with missing type arguments, Petr4 collects constraints by checking function arguments, solves those constraints, and then descends back into the arguments to insert casts where appropriate. The resulting AST contains no hidden casts or missing type arguments, which makes life easier for the interpreter.

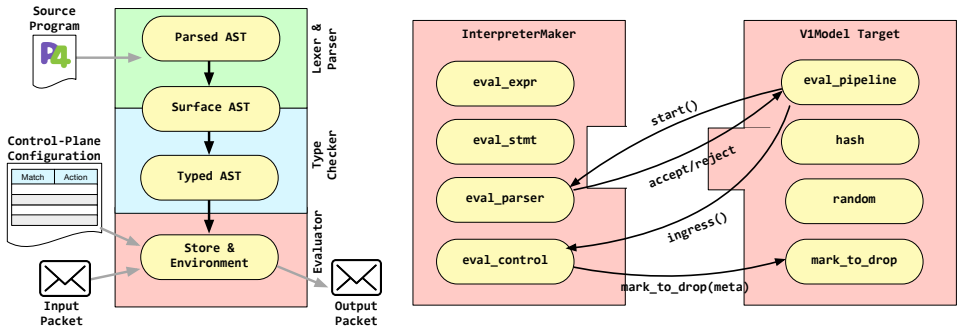



Fig. 18. P4TC implementation: (a) interpreter data flow; (b) architecture support via plug-ins.

P4 allows implicit casts between some types. For example, the variable initialization `bit<8> x = 4` will typecheck even though 4 is an `int` and not a `bit<8>`. The P4TC typechecker inserts a cast and emits a type safe initialization `bit<8> x = (bit<8>)4`. This requires changes to the inference algorithm to address the combination of implicit casts and missing type arguments, since two apparently irreconcilable constraints may become solvable with implicit casts.

P4 also includes overloading of functions and extern methods. Here the specification restricts potential type system complexity by requiring overloads to be resolvable by just looking at the number or names of arguments and not their types. Our implementation handles overloading in the code for checking function calls.

*Interpreter.* The P4TC interpreter implements a big-step evaluator, following the same basic approach as the Core P4 evaluation relation (Section 3). However, whereas Core P4 uses nondeterminism to overapproximate possible target-specific behaviors, the P4TC interpreter uses a "plugin" approach. The interpreter is an OCaml functor with the following signature:

$$\texttt{functor (T : Target)} \longrightarrow \texttt{Interpreter}$$

The `Interpreter` module signature includes functions analogous to Core P4 evaluation judgments: `eval_declaration`, `eval_statement`, and `eval_expression`.

The `Target` signature passed into the interpreter functor defines the interface between a $P4_{16}$ program and the architecture it runs on. Targets offer a list of externs:

$$\texttt{extern} : \texttt{env} \longrightarrow \texttt{state} \longrightarrow \texttt{type list} \longrightarrow (\texttt{value} \times \texttt{type})\ \texttt{list} \longrightarrow \texttt{env} \times \texttt{state} \times \texttt{value}$$

Each extern is modeled as an OCaml function that takes as input the environment (`env`), store (`state`), type arguments (`type list`), and arguments ((`value` × `type`) `list`), and returns an updated environment (`env`), updated store (`state`), and result (`value`). This expansive type reflects how $P4_{16}$ externs are allowed to do practically anything (short of modifying their caller's local variables).

Targets must also define the implementation of the packet-processing pipeline.

$$\texttt{eval\_pipeline} : \texttt{ctrl} \longrightarrow \texttt{env} \longrightarrow \texttt{state} \longrightarrow \texttt{buf} \longrightarrow \texttt{apply} \longrightarrow \texttt{state} \times \texttt{env} \times \texttt{pkt option}$$

The pipeline evaluator takes as arguments the control-plane configuration (`ctrl`), environment (`env`), store (`state`), input packet (`buf`), and a hook for interpreting parsers and controls (`apply`) and produces an updated store (`state`), environment (`env`) and output packet (`pkt option`). As can be seen from this type, P4TC does not currently support multicast, but adding it would be a relatively straightforward extension.

Figure 18 (b) shows how the `Target` and `Interpreter` pass control back and forth during execution, using the `V1Switch` architecture as a concrete example.

The output of the `InterpreterMaker` functor is an `Interpreter`, which defines a function for evaluating entire P4 programs:

$$\texttt{eval\_program} : \texttt{ctrl} \rightarrow \texttt{env} \rightarrow \texttt{state} \rightarrow \texttt{buf} \rightarrow \texttt{int} \rightarrow \texttt{prog} \rightarrow \texttt{state} \times (\texttt{buf} \times \texttt{int})\ \texttt{option}$$

It takes an initial control-plane configuration, environment, store, packet buffer and port, along with a program, and produces an updated state and (optional) modified packet as output.

We have used Petr4 to construct interpreters for two $P4_{16}$ architectures: V1Model and eBPF. V1Model is the most widely-used architecture in open-source $P4_{16}$ code. It includes a variety of features that fully exercise Petr4's interface between the interpreter and targets. The V1Model pipeline consists of 6 programmable blocks with some fixed-function components in between. The eBPF architecture supports running P4 on the Linux kernel's packet filter infrastructure. Packet filters have a simpler structure than V1Model pipelines and support a different collection of externs. These implementations show that our abstraction effectively supports multiple architectures.

Adding a new architecture to Petr4 means writing a few OCaml functions and datatypes. The implementer has to provide the function `eval_pipeline` above, which defines how control flow passes between stages of the packet-processing pipeline. The implementer must also provide data types to represent any `extern` objects provided by the architecture and implement their methods. Our current functor does not model everything left up to architectures in the specification, but it does cover the most important points. We discuss this further in *Limitations* and leave a more precise definition of architecture-dependent behavior to future work.

*Control-plane APIs.* The control plane plays an important role in the execution of most $P4_{16}$ programs by dynamically populating the match-action tables with forwarding entries. Petr4 exposes two different control-plane APIs: one based on a serialization of table entries into JSON, and the other based on the ASCII Simple Test Framework (STF) tool bundled with p4c.

For example, the following STF test checks that sending a packet containing a stack with a single `hop` header whose `port` field and `bos` fields are both `1` will cause the packet to be forwarded out on port `1`, provided the `acl` table is configured to allow the packet:

```
add MyPipe.acl MyPipe.acl.ingress_port:0 MyPipe.acl.egress_port:1 MyPipe.allow()
packet 0 03FF
expect 1 FF
```

*User interfaces.* We have equipped Petr4 with two user interfaces. The first provides a simple command-line interface for Petr4 that supports several modes of operation including parsing, type checking, and interpreting a P4 program. The second provides a web-based front-end that runs a P4 program directly in a browser, as shown in Figure 19. The web-based interface is implemented using `js_of_ocaml`, allowing Petr4 to run directly in the browser. Compared to the open-source reference implementation, which requires compiling the program with p4c to an intermediate JSON representation that can then executed on `bmv2`, a software switch, Petr4 is dramatically simpler to use. We expect that both user interfaces will be useful in teaching P4, as they eliminate much of the overhead and complexity associated with using p4c and `bmv2`—e.g., setting up virtual machines, installing dependencies, hooking into the Linux networking stack, and coordinating behavior across multiple stand-alone binaries.

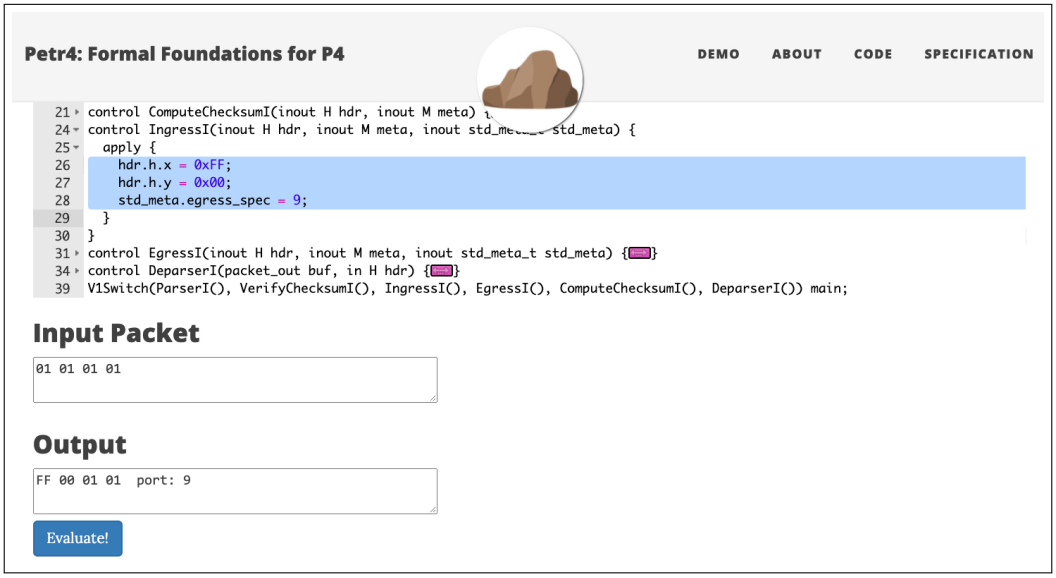



Fig. 19. Petr4 interpreter running in a web browser. The interpreter is online at cornell-netlab.github.io/petr4.

### 4.1 Limitations

Petr4 implements the vast majority of features discussed in the $P4_{16}$ specification. However, our current prototype does have some important limitations. Petr4 implements a sequential model of computation: this is more restrictive than the specification, which allows for certain forms of concurrency. Petr4 also lacks support for cloning and packet replication. Adding support for both of these features should be straightforward, but will require additional engineering both in the formalization and the implementation. Petr4 largely ignores annotations, including annotations that can affect packet processing on some architectures. Petr4 does not support `abstract` externs or user-defined initialization blocks—two recent additions to the language. Petr4's implementation of the V1Model target omits some externs, including direct-mapped objects. Finally, while the `Target` signature exposes hooks that would allow an implementation to customize behaviors left up to architectures (e.g., semantics of reads/writes to invalid headers), some behaviors have not yet been parameterized (e.g., custom properties for match-action tables).

## 5 EVALUATION

Our evaluation of Petr4 focuses on the correctness and utility of the core calculus and interpreter. We study how well they capture P4, both as it is described in the language specification, and how it is being used by the open-source community. To this end, we first explore the results of running Petr4 against the same test suite as the reference implementation. We next describe bugs and ambiguities we discovered and addressed during development, both in the reference implementation and in the language specification.

*Parser and typechecker.* We have imported 792 test cases from p4c for the parser and typechecker. These consist of "good" tests (those the typechecker should accept) and "bad" tests (those the typechecker should reject). Currently, Petr4's parser passes all 792 of these. The typechecker, on the other hand, passes 782 of these, with 10 failures due to the following issues:

- A bug in the grammar requiring an additional "lexer hack" only recently fixed in p4c.

- The `@optional` annotation for arguments, which Petr4 does not support.
- Type casts that discard significant bits of bitstrings should emit a warning, but do not have to fail. p4c's suite expects them to fail.
- p4c rejects programs that shadow names (control-plane and local scope) of functions, actions, controls, tables, and parsers, whereas Petr4 is more permissive. The specification says the compiler "may provide a warning if multiple resolutions are possible for the same name" for some situations but does not require it to be a type error.
- Implicit casts from signed to unsigned integers may turn a bad (negative) operand for division into a good (positive) value. Division with negative values is not allowed by the specification, so the difference with p4c in this case is only because Petr4 checks the sign after doing implicit casts rather than before.
- Several restrictions on the structure of programs are imposed by `V1Model` but not enforced by Petr4. For example, `V1Model` requires deparser code to be free of conditionals, but Petr4 does not enforce this kind of architecture-specific syntactic restriction yet.

There are an additional 110 tests imported from p4c which are unsupported by our typechecker. For more detail see Section 4.1 above.

*Interpreter.* Of the good checker tests, 121 are accompanied by corresponding STF files used to test the correctness of p4c's back end. As described in section 4, our control plane API allows us to run these same tests on our interpreter. We currently pass 95 of these STF tests with 26 failures. Most of our failures (20 tests) are P4 programs written in architectures unimplemented by Petr4 (`PSA` and `UBPF`). The remaining 6 utilize externs in the `EBPF` and `V1Model` architectures that Petr4 also leaves unsupported, such as multicast and the `crc16` checksum algorithm. Some of the more interesting tests imported from p4c are described in detail in Figure 20. We also provide 40 of our own custom STF tests accumulated during test-driven development of the interpreter that address difficult edge cases of the language we felt the p4c suite did not sufficiently exercise. Petr4 passes all 40 custom tests, a sample of which are described in detail in Figure 21.

In developing Petr4, we uncovered bugs in p4c, ambiguities in the informal $P4_{16}$ spec, and issues with p4c arising from choices it made to resolve these ambiguities. We describe some bugs here, but see Figure 22 and Figure 23 for a full list. All bugs have been reported to either the $P4_{16}$ specification repository or the p4c repository on Github.

*Grammar and parser.* The $P4_{16}$ grammar allowed annotations to either take an expression list or a list of key-value pairs for their arguments. This approach introduced an ambiguity into the grammar: there was no way to discern whether an empty list was an expression list or a key-value pair list. We eliminated this ambiguity by allowing the annotations to take a non-terminal in the grammar called `argumentList` in which each argument could be either an expression or a key-value pair. Additionally, this simplification allowed for more flexible behavior—mixing expression and key-value pair arguments—while still supporting the old behavior.

Even before support for top-level functions was implemented, we discovered a conflict between function declarations and newtype declarations. (A newtype declaration `type name_t old_t` creates an opaque type alias `name_t` for the type `old_t`, like `newtype` in Haskell.) The P4 grammar begins both function and newtype declarations with a token sequence `TYPE TYPE_IDENTIFIER`. The `TYPE` token corresponded to not only a `nonTypeName` in the function declaration, but also the `type` keyword in the newtype declaration. Thus, the parser could either reduce a `nonTypeName` out of `TYPE` or shift to recognize a newtype declaration.

*Type checker.* We found multiple discrepancies between p4c and the $P4_{16}$ specification with respect to typechecking. p4c rejected any program with headers containing multiple `varbit` fields.

| Test File | Description (Features Tested) | LoC | headers (#, bits) | parser states | tables? |
|---|---|---|---|---|---|
| issue2287-bmv2.p4 | apply binary operators to function calls with side-effects (operators, side-effects, copy-in/copy-out) | 95 | (3, 248) | 1 | ✗ |
| enum-bmv2.p4 | equality test on basic enum (enums) | 44 | (1, 96) | 1 | ✗ |
| issue1025-bmv2.p4 | call `lookahead` as argument to `extract` (`extract`, `lookahead`, variable-size bitstrings) | 176 | (3, 468) | 3 | ✗ |
| subparser-with-header-stack-bmv2.p4 | subparser invocation while parsing a header stack (header stacks, parser application) | 168 | (7, 224) | 5 | ✗ |
| test-parserinvalidargument-error-bmv2.p4 | variable-size extract triggers parser error (variable-size bitstrings, parser errors, control-flow) | 118 | (2, 128) | 2 | ✗ |
| table-entries-priority-bmv2.p4 | `priority` annotation affects constant table entries (table application, priority, constant table entries, `ternary`) | 89 | (1, 48) | 1 | ✓ |
| default_action-bmv2.p4 | table application falls through to non-trivial default action (table application, default action, control-plane interface) | 35 | (1, 64) | 1 | ✓ |
| table-entires-ser-enum-bmv2.p4 | serializable enum appears in constant table entries (serializable enums, table application, constant table entries) | 85 | (1, 16) | 1 | ✓ |
| checksum3-bmv2.p4 | compute checksum using `csum16` (externs) | 195 | (3, 320) | 3 | ✗ |
| count_ebpf.p4 | stateful extern from `ebpf_model` architecture (stateful externs, target abstraction) | 62 | (2, 272) | 2 | ✗ |

Fig. 20. Selection from P4C's STF test suite.

| Test File | Description (Features Tested) | LoC | headers (#, bits) | parser states | tables? |
|---|---|---|---|---|---|
| bitstrings.p4 | emit results of binary operators on bitstrings (bit-strings, `emit`) | 97 | (0, 0) | 1 | ✗ |
| stack.p4 | complex operations on header stacks (header stacks) | 141 | (43, 688) | 1 | ✗ |
| union.p4 | complex operations on header unions (header unions) | 130 | (6, 72) | 1 | ✗ |
| scope.p4 | function name shadowing (lexical scope) | 52 | (1, 8) | 1 | ✗ |
| error2.p4 | triggers parser errors (parser errors, control-flow) | 98 | (2, 32) | 2 | ✗ |
| subparser.p4 | direct application of sub-parser from main parser (parser application, `verify`, control-flow) | 133 | (5, 40) | 7 | ✗ |
| exit.p4 | `exit` statement in nested calls to actions (control-flow) | 107 | (13, 104) | 3 | ✗ |
| subcontrol.p4 | direct application of sub-control with `exit` from egress processing (control application, control-flow) | 71 | (2, 16) | 1 | ✗ |
| table.p4 | apply control-plane-defined table (control-plane interface, table application) | 65 | (1, 8) | 1 | ✓ |
| table3.p4 | apply table with constant `lpm` and `ternary` entries (constant table entries, `lpm`, `ternary`, table application) | 94 | (1, 8) | 1 | ✓ |
| switch-stmt.p4 | `switch` statement on table with constant entries (constant table entries, table application, `switch` statement) | 93 | (2, 16) | 2 | ✓ |

Fig. 21. Selection from PETR4's custom STF test suite.

However, the specification only requires that such headers cannot be used in `extract`. Since P4 implementations are allowed to provide extensions that could make such headers useful, P4C's restriction was too strong compared to the spec. Conversely, the specification is too restrictive in permitting division and modulo between positive `int` values only, whereas P4C relaxes this constraint to permit the same operations between `bit` values.

| Category | Issues Description |
|---|---|
| Grammar and Parser | (a) The parser had a conflict with the `TYPE` token where it could either reduce a `nonTypeName` out of `TYPE` or shift to recognize a newtype declaration. This problem was detected before it could manifest in the compiler since at the time, top-level functions were not yet implemented.<br>(b) The parser incorrectly resolved names with a "dot" using the local context instead of the top-level context.<br>(c) The parser rejected actions with dot prefix. |
| Typing | (d) The type system was sometimes nominal and sometimes structural. The behavior was not consistent across individual programming constructs.<br>(e) The type of a list expression was a tuple which inadvertently allowed tuples to be assigned to structs since list expressions are allowed to be assigned to structs.<br>(f) The front-end's constant folding transformed a program that was not well-typed into one that was.<br>(g) Tuples in set contexts are inconsistently flattened when checking matches against keys. |
| Other | (h) The compiler did not clearly enforce the constraint that the `default_action` must appear after `actions`.<br>(i) The compiler did not clearly enforce that values of type `int` should all be compile-time known values.<br>(j) An STF test had two lines uncommented that were supposed to be commented.<br>(k) The compiler rejected any program with headers containing multiple `varbit` fields even though the spec only states such headers cannot be used in `extract`.<br>(l) The compiler did not stop compiling after encountering an error. |

Fig. 22. p4c Issues

| Category | Issues Description |
|---|---|
| Syntax | (a) Annotations could take either an expression list or a keyValuePair list for their arguments but this made the grammar ambiguous because there was no way of telling whether the empty list was an expression list or keyValuePair list. This issue came to light before free-form annotations were added.<br>(b) The $P4_{16}$ grammar required all optional type parameters to be non-type names even though there were use cases that contradicted this restriction and the compiler did not impose such a restriction.<br>(c) The spec does not impose a requirement on the placement of table entries even though it seems like it should so that a typechecker can process the properties in order. |
| Types | (d) The spec stated that the compiler does not insert implicit casts for the arguments to methods or functions. This was an undesirable restriction.<br>(e) The spec is too restrictive because it only permits division and modulo between positive `int` values.<br>(f) The spec does not explicitly allow header unions in header stacks.<br>(g) The spec does not clarify whether int values can be cast to bool or not. It also does not state whether assigning a bit value to an int variable is allowed by implicitly casting the value to int. |
| Operational | (h) The spec did not define the concatenation operator's behavior on signed and unsigned bitstrings. |

Fig. 23. $P4_{16}$ Specification Issues

*Semantics of P4 constructs.* The P4 specification originally imposed a restriction on inserting implicit casts for the arguments to methods or functions. Implicit casts were intended to reduce the friction for the programmer and allow her to use constants naturally. Thus, the restriction was rather undesirable and was lifted when the specification was amended to allow implicit casts on `in` arguments for methods and functions. We also influenced another amendment to define explicitly the concatenation operator's behavior for both signed and unsigned bitstrings. Prior to this change, it was unclear whether the concatenation operator was supported for signed bitstrings until we found that p4c allowed it.

*Inconsistent type system, buggy inference, untyped constant folding.* These bugs emphasize subtle ways in which handling types can be tricky to get right. First, we discovered that the type system

was inconsistent in that it was sometimes nominal and sometimes structural. For example, controls were structural in architecture definitions, and nominal elsewhere. The solution we implemented was to distinguish a control from the type of the control. Consequently, a control could no longer be used as the type of a parameter. In conjunction, a tuple cannot be assigned to a struct where list expressions can, and in fact have a tuple type. This subtlety allowed tuples to be assigned to structs. This was fixed by checking for a tuple type by means of a struct-like type conversion and introducing a new type internally for list expressions. Another example of a subtle type issue was in P4C's constant folding. Because it was not paying enough attention to types, it transformed an ill-typed program into a well-typed one in some cases.

## 6 CASE STUDY: ADDING TYPE-SAFE UNIONS TO P4

In this section, we exercise our formal semantics by adding union types to P4. Uncertainty about the safety of language extensions is a perennial concern for the P4 Language Design Working Group, resulting in language features which are hamstrung or, worse yet, buggy. With formal semantics, we can prove adding a feature is safe by defining and proving correct a translation from the augmented language back into the original one.

P4 already has a restricted form of unions, but they can only contain header types and lack a type-safe elimination form. To address this shortcoming, we define an extension of P4 with tagged unions and formalize its semantics. We then define a translation from P4 with unions into standard P4 and prove that the translation preserves program semantics.

*Syntax.* We extended the syntax to allow the declaration of a union type. We allow assignment to union fields, and extended the statements with a switch statement where cases are either union fields or default. Finally, we add union values which consist of the union type, its "active" field, and that field's value.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mathsf{union}\ X\ \{\overline{\tau\ f}\} \\ stmt & ::= & \ldots \mid \mathsf{switch}\ (exp)\ \{\overline{lbl\colon \{\overline{stmt}\}}\} \\ lbl & ::= & \ldots \mid \mathsf{default} \\ val & ::= & \ldots \mid X\{f, val\} \end{array}$$

*Typing rules.* We need two new typing rules for statements that include unions (as well as an auxiliary judgment switchaseok to check the branches—see the appendix for details).

$$\textsc{T-Union}\ \frac{\Sigma, \Gamma, \Delta \vdash exp_1 : \mathsf{union}\ X\ \{\overline{\tau\ f}\} \quad \Sigma, \Gamma, \Delta \vdash exp_2 : \tau_i}{\Sigma, \Gamma, \Delta \vdash exp_1.f_i = exp_2 \dashv \Sigma, \Gamma}$$

$$\textsc{T-Switch}\ \frac{\Sigma, \Gamma, \Delta \vdash exp : \mathsf{union}\ X\ \{\overline{\tau\ f}\} \quad \overline{lbl \in \{\overline{f}, \mathsf{default}\}} \quad \Sigma, \Gamma, \Delta \vdash \mathsf{switchcaseok}(\overline{\tau\ f}, \overline{lbl\colon \{\overline{stmt}\}})}{\Sigma, \Gamma, \Delta \vdash \mathsf{switch}\ (exp)\ \{\overline{lbl\colon \{\overline{stmt}\}}\} \dashv \Sigma, \Gamma}$$

*Evaluation rules.* Union variables are initialized upon declaration ($\mathsf{init}_\Delta\ X = X\{f_0, \mathsf{init}_\Delta\ \tau_0\}$). A union's value is modified by assigning a value to one of its fields:

$$\textsc{E-Union}\ \frac{\langle C, \Delta, \sigma, \epsilon, exp_1\rangle \Downarrow_{lval} \langle \sigma_1, lval\rangle \quad \langle C, \Delta, \sigma_1, \epsilon, exp_2\rangle \Downarrow \langle \sigma_2, val\rangle \quad \langle \sigma_2, \epsilon, lval.f := val\rangle \Downarrow_{write} \sigma_3}{\langle C, \Delta, \sigma, \epsilon, exp_1.f_i := exp_2\rangle \Downarrow \langle \sigma_3, \epsilon, \mathsf{cont}\rangle}$$

To evaluate a union switch statement on a union with value $X\{f_i, val_i\}$, we match $f_i$ against the labels. If it matches a label other than default, we evaluate the corresponding block in an environment where $f_i$ maps to $val_i$ (E-UnionSwitch). Note that we can rewrite the blocks so that they have no local variable declaration with the same name as their corresponding label, so we

don't violate our naming convention. If a default is provided and $f_i$ matches no other label, we proceed with evaluating the corresponding block in the same environment. If no default is provided and there is no match, we skip.

$$\textsc{E-UnionSwitch}\quad \frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle \sigma_1, X\{f_i, val_i\}\rangle \\ \text{match\_union\_case } (\overline{lbl\colon \{\overline{stmt}\}}, f_i) = (f_i, k) \qquad \ell \text{ fresh} \\ \sigma_2 = \sigma_1[\ell \mapsto val_i] \qquad \langle C, \Delta, \sigma_2, \epsilon[f_i \mapsto \ell], \overline{\{stmt_k\}}\rangle \Downarrow \langle \sigma_3, \epsilon', sig\rangle\end{array}}{\langle C, \Delta, \sigma, \epsilon, \text{switch } (exp)\ \{\overline{lbl\colon \{\overline{stmt}\}}\}\rangle \Downarrow \langle \sigma_3, \epsilon, sig\rangle}$$

*Translation to standard P4.* We use records in standard P4 to implement unions. The mapping $[\![\cdot]\!]$ translates from P4 extended with unions to P4. Expressions are not changed in the extended language. Declaring a new union type translates to a typedef:

$$[\![\text{union } X\ \{\overline{\tau\ f}\}]\!] = \text{typedef } \{tag : \text{bit}\langle n\rangle, \overline{f : \tau}\}\ X$$

Here, *tag* keeps track of the "active" union field. Declaring a new variable of the union type translates to declaring a new variable of the corresponding record type.

$$[\![X\ x]\!] = X\ x, x.tag := 0, \overline{x.f_i := \text{init}_\Delta\ \tau_i}$$

The only extensions to statements are assignment to union fields, and the union switch. Thus, except for the following cases, statements are translated homomorphically. In the clause for assignment the field names $\overline{f_j}$ range over all $\overline{f}$ except $f_i$.

$$\begin{aligned}[\![exp_1.f_i := exp_2]\!] &= exp_1 := \{tag : \text{bit}\langle n\rangle = i, f_i : \tau_i = exp_2, \overline{f_j : \tau_j = \text{init}_\Delta\ \tau_j}\} \\ [\![\text{switch } (exp)\ \{\overline{lbl\colon \{\overline{stmt}\}}\}]\!] &= X\ tmp := exp; \\ &\quad \text{if } (c_0)\ \{b_0\} \ldots \text{else if } (c_n)\ \{b_n\} \text{ else } \{\}\end{aligned}$$

where $(c_i, b_i) = \text{trans\_union\_case } (lbl_i, \overline{stmt}_i)$ and:

$$\begin{aligned}\text{trans\_union\_case } (\text{default}, \overline{stmt}) &= (\text{true}, [\![\overline{stmt}]\!]) \\ \text{trans\_union\_case } (f_j, \overline{stmt}) &= (tmp.tag == j, \tau_j\ f_j := tmp.f_j, [\![\overline{stmt}]\!])\end{aligned}$$

Note that *tmp* is a fresh variable name. Union values are translated to records:

$$[\![X\{f_i, val_i\}]\!] = \{tag : \text{bit}\langle n\rangle = i, f_i : \tau_i = val_i, f_j : \tau_j = \text{init}_\Delta\ \tau_j\}$$

For records, headers, and header stacks that are inductively built from other values, we have:

$$\begin{aligned}[\![\{\overline{f : \tau = val}\}]\!] &= \{\overline{f : \tau = [\![val]\!]}\} \\ [\![\text{header } \{\text{valid}, \overline{f : \tau = val}\}]\!] &= \text{header } \{\text{valid}, \overline{f : \tau = [\![val]\!]}\} \\ [\![\text{stack } \tau\ \{\overline{val}\}]\!] &= \text{stack } \tau\ \{\overline{[\![val]\!]}\}\end{aligned}$$

All other values are translated homomorphically. We translate stores by translating their range: $[\![\sigma]\!]$ has the same domain as $\sigma$. If $\sigma(l) = val$, then $[\![\sigma]\!](l) = [\![val]\!]$.

*Translation property.* We prove in the appendix [Doenges et al. 2020] that the translation function is semantics-preserving. Specifically, we prove the following theorem by induction on the statement evaluation rules, and a case analysis on the last rule in the derivation.

Theorem 6.1. *If $\langle C, \Delta, \sigma, \epsilon, stmt\rangle \Downarrow \langle \sigma', \epsilon', sig\rangle$, then $\langle C, \Delta, [\![\sigma]\!], \epsilon, [\![stmt]\!]\rangle \Downarrow \langle \sigma_t, \epsilon_t, sig\rangle$ and $\langle [\![\sigma']\!], \epsilon'\rangle \subseteq_{env} \langle \sigma_t, \epsilon_t\rangle$. We say $\langle \sigma_1, \epsilon_1\rangle \subseteq_{env} \langle \sigma_2, \epsilon_2\rangle$ if $\epsilon_1$'s domain is a subset of $\epsilon_2$'s domain, and for all lval in $\epsilon_1$'s domain, if $\epsilon_1(lval) = \ell_1$ and $\sigma_1(\ell_1) = val$, then $\epsilon_2(lval) = \ell_2$ and $\sigma_2(\ell_2) = val$.*

## 7 RELATED WORK

The problem of formalizing the semantics of a language is one of the oldest problems in our field, and it remains an active and relevant area of research today. This section briefly reviews some of the most closely related work.

*Semantics for industry languages.* Formal models have recently been developed for a growing number of practical languages used in industry. Pioneering work by Milner, Tofte, Harper, and MacQueen developed a formal definition of Standard ML, one of the first languages to be given such a treatment [Milner et al. 1997]. More recently, a number of prominent efforts have developed semantics for languages as complex and diverse as JavaScript [Guha et al. 2010; Park et al. 2015], WebAssembly [Haas et al. 2017], C [Leroy 2009], x86-TSO [Sewell et al. 2010a], and the POSIX shell [Greenberg and Blatt 2020]. Like Petr4, these efforts build on decades of foundational work in semantics [Kahn 1987; Plotkin 1981; Scott and Strachey 1971] and semantics engineering [Sewell et al. 2010b]. Recent work by Ruffy et al. has profitably combined fuzzing and translation validation to find numerous bugs in p4c [Ruffy et al. 2020]. Their Gauntlet translation validator defines the behavior of P4 programs by an SMT-LIB encoding, making program equivalence checkable with a single Z3 query. Our work focused on building a reusable semantics which could, for example, verify the translation used in Gauntlet. Of course, the translation in Gauntlet could also be productively applied to fuzzing the Petr4 interpreter.

*Semantics for networks.* In the networking context, Sewell et al. developed mechanized formal models of TCP and UDP [Bishop et al. 2018] using HOL4. A key challenge was designing a "loose" semantics that could accommodate the implementation choices made by different network stacks. The same issue arises in Petr4 when modeling architecture-specific features, such as read and write operations to invalid headers. Guha, Reitblatt, and Foster developed a verified compiler from NetCore, a high-level policy language, to OpenFlow, an early software-defined networking standard [Guha et al. 2013]. Another line of recent work has focused on eBPF, the packet-processing framework supported in the Linux kernel. The JitK compiler [Wang et al. 2014] uses a machine-verified just-in-time compiler to generate code that is guaranteed to satisfy the safety conditions enforced by the kernel verifier, while JitSynth leverages program synthesis [Geffen et al. 2020].

*Network verification.* As mentioned in Section 1, there is a growing body of work focused on data plane and control plane verification, including Header Space Analysis [Kazemian et al. 2012], Anteater [Mai et al. 2011], NetKAT [Anderson et al. 2014], Batfish [Fogel et al. 2015], ARC [Gember-Jacobson et al. 2016], and Minesweeper [Becket et al. 2017], to name a few. Other tools have applied techniques such as predicate transformers [Liu et al. 2018], symbolic execution [Nötzli et al. 2018; Stoenescu et al. 2018], or translation into another language, such as Datalog [McKeown et al. 2016], to verify P4 programs. However, none of these tools are based on a foundational semantics like Petr4—they either rely on ad hoc models or rely on an existing implementation such as p4c. Kheradmand and Rosu developed an operational model for P4 in the K framework [Kheradmand and Rosu 2018]. The P4K project implemented $P4_{14}$, which has substantially different syntax and semantics from $P4_{16}$, and provided an interpreter without an accompanying type system. The interpreter is implemented in the K framework, which was able to produce verification and translation validation tools automatically from the interpreter definition. However, the encoding in K limits its reusability outside of the K framework.

## 8 CONCLUSION AND FUTURE WORK

This paper introduced Petr4, a formal framework that models the semantics of P4. We developed a clean-slate definitional interpreter for P4 as well as a formal calculus that models the essential

features of the language. The implementation has been validated against over 750 tests from the reference implementation and the calculus proven to satisfy type-safe termination.

In the future, we would like to extend our calculus to model the full language and publish it as the official specification of the language for the P4 community. We believe it would be a valuable resource for designers, compiler writers, and application programmers alike. Concretely, we would like to close the gap between our definitional interpreter and calculus, obtaining a formal semantics that covers the entire language. It would also be attractive to have a mechanized semantics so the reference interpreter can be extracted from the formalization. Toward this end, we have begun porting our definitional interpreter to Coq. We do not foresee any major technical challenges and believe it should be possible to complete this task quickly though porting our type soundess and termination proofs will take longer. The biggest obstacles are likely to be related to architectures and extern functions, which are straightforward to handle in principle but somewhat tedious to implement in practice. Looking further ahead, we eventually hope to use our Coq formalization to develop a verified compiler for P4. We are also interested in using Petr4 to guide development of further enhancements to P4—e.g., designing a smaller core language to streamline development of tools, and adding full support for generics and a module system to the language.

## ACKNOWLEDGMENTS

We are grateful to the POPL'21 reviewers for their feedback and suggestions for improving this paper. We wish to thank Chris Sommers for many discussions on formalizing P4 and Michael Greenberg for advice on presenting this work, and we wish to especially thank our many colleagues in and out of the Cornell Netlab who gave generous and detailed feedback on drafts. Our work has been supported in part by the National Science Foundation under grant FMiTF-1918396, the Defense Advanced Research Projects Agency (DARPA) under Contract HR001120C0107, and gifts from Keysight, Fujitsu, and InfoSys.

## REFERENCES

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM POPL*. 113–126. https://doi.org/10.1145/2535838.2535862

Ryan Becket, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM*. 155–168. https://doi.org/10.1145/3098822.3098834

Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. 2018. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *JACM* 66, 1 (Dec. 2018), 1:1–1:77. https://doi.org/10.1145/3243650

Nikolaj Bjorner and Karthick Jayaraman. 2015. Checking Cloud Contracts in Microsoft Azure. In *ICDCIT*. Springer-Verlag, 21–32. https://doi.org/10.1007/978-3-319-14977-6_2

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

Cisco Systems. 2018. Cisco DNA Analytics and Assurance. Available at https://www.cisco.com/c/en/us/solutions/enterprise-networks/dna-analytics-assurance.html.

Luis Damas. 1984. *Type Assignment in Programming Languages*. Ph.D. Dissertation. University of Edinburgh. Available at http://hdl.handle.net/1842/13555.

Catherine Dodge and Stephen Quigg. 2018. A Simpler Way to Assess the Network Exposure of EC2 Instances: AWS Releases New Network Reachability Assessments in Amazon Inspector. Archived at https://web.archive.org/web/https://aws.amazon.com/blogs/security/amazon-inspector-assess-network-exposure-ec2-instances-aws-network-reachability-assessments/.

Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2020. Petr4: Formal Foundations for P4 Data Planes. arXiv:2011.05948 [cs.PL]

A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*. 469–483.

Nate Foster. 2019. Type error due to inference/substitution? Github bug report. Archived at https://web.archive.org/web/https://github.com/p4lang/p4c/issues/2036.

Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *CAV*. https://doi.org/10.1007/978-3-030-53291-8_29

Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*. 300–313. https://doi.org/10.1145/2934872.2934876

Michael Greenberg and Austin J. Blatt. 2020. Executable Formal Semantics for the POSIX Shell. In *POPL*. https://doi.org/10.1145/3371111

Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-Verified Network Controllers. In *PLDI*. 483–494.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP*. https://doi.org/10.1007/978-3-642-14107-2_7

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *PLDI*. 185–200. https://doi.org/10.1145/3062341.3062363

Stefan Heule, Konstantin Weitz, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. 2019. Leveraging P4 to Automatically Validate Networking Switches. Presentation at ONF Connect. Slides available at https://www.opennetworking.org/wp-content/uploads/2019/09/2.30pm-Stefan-Heule-P4-Presentation.pdf.

Mukesh Hira and LJ Wobker. 2015. Improving Network Monitoring and Management with Programmable Data Planes. P4 Language Consortium Blog. Available at https://p4.org/p4/inband-network-telemetry/.

Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*. 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. 121–136. https://doi.org/10.1145/3132747.3132764

Jacques-Henri Jourdan and François Pottier. 2017. A Simple, Possibly Correct LR Parser for C11. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 4 (2017), 1–36. https://doi.org/10.1145/3064848

Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS)*. Springer-Verlag, 22–39. https://doi.org/10.1007/BFb0039592

Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*. 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

Ali Kheradmand and Grigore Rosu. 2018. P4K: A Formal Semantics of P4 and Applications. (2018). arXiv:1804.01468 [cs.NI]

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *SOSP*. 599–613. https://doi.org/10.1145/3132747.3132759

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 *ACM SIGCOMM*. 490–503. https://doi.org/10.1145/3230543.3230582

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *SIGCOMM*. 290–301. https://doi.org/10.1145/2018436.2018470

Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. *Automatically Verifying Reachability and Well-Formedness in P4 Networks.* Technical Report MSR-TR-2016-65. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/09/p4nod.pdf

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. p4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SOSR*. 5:1–5:7. https://doi.org/10.1145/3185467.3185497

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*. 346–356. https://doi.org/10.1145/2737924.2737991

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

Gordon D Plotkin. 1981. A Structural Approach to Operational Semantics. (1981).

Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *OSDI*. https://www.usenix.org/conference/osdi20/presentation/ruffy

Dana Scott and Christopher Strachey. 1971. *Toward a Mathematical Semantics for Computer Languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group Oxford.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010a. x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/

1785414.1785443

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010b. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.* 20, 1 (Jan. 2010), 71–122. https://doi.org/10.1017/S0956796809990293

Christian Skalka, John Ring, David Darias, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. 2019. Proof Carrying Network Code. In *ACM CCS*. 1115–1129. https://doi.org/10.1145/3319535.3363214

Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *SIGCOMM*. https://doi.org/10.1145/3230543.3230548

Aldo Svaldi. 2019. A Single Network Card Caused CenturyLink's Nationwide Outage. The Denver Post. Archived at https://web.archive.org/web/20190202225936/https://www.denverpost.com/2019/01/11/centurylink-network-outage-denver/.

The P4 Language Consortium. 2018. *P4 Language Specification, Version 1.1.0.* Available at https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems*. 1–7. https://doi.org/10.1145/2349896.2349905

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*. 33–47. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi