# PROGRAMMING LANGUAGE FOUNDATIONS FOR PACKET PROCESSING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ryan Doenges

August 2023

PROGRAMMING LANGUAGE FOUNDATIONS FOR PACKET PROCESSING

Ryan Doenges, Ph.D.

Cornell University 2023

This dissertation gives semantics to P4, a domain-specific language for describing packet processing in packet-switched computer networks. Additionally it describes verification tools for checking the equivalence of P4 programs. These verifiers can be used to check that a P4 compiler has not introduced bugs into programs while optimizing them. The verification methodology combines manual proof in an LCF-style proof assistant with automatic decision procedures that rely on SAT/SMT solvers for a compact trusted computing base.

## BIOGRAPHICAL SKETCH

Ryan Doenges was born in Seattle. He received a Bachelor of Science in Mathematics from the University of Washington in 2017.

This thesis is dedicated to my friends who left.

## ACKNOWLEDGEMENTS

Chapters 3 and 4 were adapted from prior publications with the permission of my co-authors [37, 38]. My research was funded in part by the National Science Foundation under grant FMiTF-1918396, the Defense Advanced Research Projects Agency (DARPA) under Contract HR001120C0107, and gifts from Fujitsu, Google, InfoSys, and Keysight.

Much of the work that lead to this thesis was carried out under exceptional conditions between 2020 and 2022 due to the COVID-19 pandemic. I am grateful to my colleagues for finding a way to make progress on our research in that period. I am also grateful to Rolph Recto and Shir Maimon, who shared an apartment with me at the time and helped me understand graduate school life even as it melted into air.

I thank the members of the PLDG seminar for many fruitful discussions and insighful comments over the years. I thank my officemates in the Netlab, particularly Parisa Atei, Eric Campbell, Yunhe Liu, Hussain Ahmad, and Mark Moeller, for all the advice, company, and laughter. Without the past (and continuing) support of my undergraduate advisor Zach Tatlock and mentors James Wilcox and Doug Woos I would not have finished this thesis. Most of all, I thank my advisor Nate Foster for his patience, his understanding, and his perspective.

I thank my family back in the state of Washington for their support and faith even when I was a real grouch.

I thank Cal for his love, companionship, and great hair. I thank him for helping me write talks by telling me when my extemporizing starts to click. The argument that begins Chapter 1 was originally developed for my defense talk using this method. I thank him for his forceful interventions on divers topics: whether I've been working hard or hardly working (the former), whether I have struck the right tone in a draft of an email (usually yes), whether I ought to take a break (yes). I thank him for knitting me a hat and hand warmers so I could think and type in the winter.

**TABLE OF CONTENTS**

## LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

> You have seen him spout; then declare what the spout is; can you not tell water from air?
>
> ---
>
> Moby-Dick

Many problems arising in language design are the result of languages growing beyond any individual's capacity for reasoning. Modern kitchen-sink approaches to language design only intensify these issues. In response, carefully curated domain-specific languages (DSLs) have emerged as a simpler way to express computations arising in limited classes of applications. With less going on at the semantic level, a program written in a DSL is easier to mechanically verify, optimize, and translate than a program written in a general-purpose language.

The same may be said about computer architecture. If an instruction set is too capable and general purpose, realizing it in hardware can require a lot of complex and power-hungry logic like branch predictors or multi-level caches. In a domain-specific architecture (DSA), everything from instruction decoding to the memory hierarchy can be designed to fit the needs of the domain. This can save power and improve performance but in general requires the adoption of a new programming model, i.e., a DSL. The term "domain-specific architecture" was popularized by the 2018 Turing Award lecture of Hennessey and Patterson, from which we might attempt to glean the state of the art in language design for these architectures. The laureates listed several promising DSLs: Matlab [59], TensorFlow [1, 2], P4 [16], and Halide [91]. These are domain-specific languages designed for scientific computing, machine learning, networking, and image processing respectively; with the exception of Matlab each

language restricts the structure and control flow of programs to fit the demands of its domain and associated DSAs.

For instance, the P4 language is designed for packet processing and targets packet processing pipelines[1], which are DSAs that manipulate network packets one at a time in a sequence of pipelined stages. If one stage takes too long to execute, the entire pipeline has to stall until it finishes, so architectures do not permit I/O or iteration within the pipeline. Keeping the pipeline running smoothly and predictably keeps packets from being dropped or delayed, which is important for network-wide performance. At the language level, this domain-specific restriction is enforced by the P4 DSL, which prohibits looping by omitting loops from its syntax and prohibits recursion by type checking. As a result all P4 programs terminate and the language is not Turing complete. This is a significant semantic difference from ISWIM-like languages, which are Turing-complete no matter how you instantiate them. Constraints in machine learning accelerators and GPUs have lead to similar language-level restrictions in the TensorFlow and Halide DSLs.

This trend towards restricted DSLs makes the systems built on top of them harder to understand and trust. Mature languages like ML and C have a lot of tooling built up around them, including high assurance verification and compilation toolchains like the Verified Software Toolchain [6] (backed by the CompCert verified compiler [70]) and the CakeML Characteristic Formulae verifier [47] (backed by the CakeML verified compiler [69]). Because restricted DSLs with new target architectures do not fit nicely into the front end or back end of these compilers, they require mostly new compiler infrastructure, metatheory, and verification frameworks.

In the case of P4, some work has been done to validate compiler optimizations in the Gauntlet translation validator [93]. A translation validator checks that a particular

[1]The name “P4” abbreviates “Programming Protocol-Independent Packet Processors” [16].

program has been optimized soundly by some third-party compiler. Gauntlet, like many modern verification tools, reduces the problem of validation to logical queries which are answered by an off-the-shelf satisfiability modulo theories (SMT) solver. This nicely modularizes the problem into verification condition (VC) generation and VC solving, but as a general approach to verification it can be unconvincing due to its large trusted computing base.

Given a software system $s$ equipped with a specification $\varphi$, a trusted computing base (TCB) of $s$ is a set of parts of $s$ which must function correctly in order for $s$ to satisfy the specification $\varphi$. The definite article is typically used, e.g., "the TCB of $s$ is compact," but this can be misleading. Depending on the specification chosen for $s$, a TCB may be a small fraction of its parts or encompass the entire system. Verifying portions of the TCB is often said to remove parts from the TCB, but in the fine print one finds that it was only exchanged. Consider verifying a trusted component $c$ of $s$ with a verifier $v$. If $v$ reports that $c$ obeys its specification, we may remove $c$ from the TCB, but at the price of incorporating the TCB of $v$. This law was popularized by the "trusting trust" Turing Award lecture of Thompson [105], although that lecture discusses compilers and not verifiers.

Verifying a compiler with solver-aided translation validation exchanges trust in the compiler for trust in the VC generator and solver. A VC generator is really a *logical compiler*, that is, a VC generator compiles programs into logical formulae or SMT-LIB [10] commands. Like any compiler targeting a performance-sensitive programming language, a VC generator must carefully optimize and encode source programs into its target language. This introduces complexity into the VC generator itself which is comparable to the complexity of the compiler it validates. In light of this, it is unclear how much solver-aided validation actually improves a TCB.

Achieving high assurance in new DSL compilers will require creative approaches

to verification that can balance the the benefits of TCB minimization with the costs of developing verifiers. In this thesis we introduce *verified solver-aided translation validation*, which strikes a new balance in the area suited to the needs of modern DSLs, and show its applicability to sublanguages of P4. We achieve a TCB comparable to traditional verified compiler TCBs but benefit from modern SMT solvers in the verifier proper instead of performing laborious tactic-based reasoning, which is reserved for meta-level reasoning about the soundness of the verifier itself. For modern DSLs like P4 with restricted computational models and compact feature sets, verifying a solver-aided validator is a better choice than either verifying a regular compiler or verifying an unaided validator.

In the remainder of this chapter I provide a short introduction to programmable networking, describe the state of the art in verified compilers, and summarize the contributions of this thesis.

## 1.1 Programmable Networking

By tradition the data plane, the part of a computer network responsible for packet forwarding, implements standardized protocols directly in hardware. Data plane devices like switches and network interface cards (NICs) expose some form of configurability to network operators, who configure devices to implement network-wide objectives. With a fixed collection of protocols to support and a limited configuration interface, networking hardware has grown highly specialized, stable, and efficient.

In other words, the stability of existing data planes is due to their inflexibility. Efforts to improve the flexibility of data planes have expanded their configuration interface and ultimately introduced true programming to data planes. In these programmable data planes, forwarding is specified from beginning to end in a domain specific language (DSL) such as the P4 programming language. These programmable

data planes, despite their programmability, perform on par with existing fixed-function data planes. However, their flexibility makes them less stable and less trustworthy than fixed-function devices.

Some of the risk arises from the introduction of new protocols. Standardized protocols such as Ethernet are well understood. In traditional networking, verifying a new Ethernet device is the same problem every time—so much so that there are firms that sell automated Ethernet test harnesses. These boxes connect to an Ethernet device via its ordinary Ethernet ports and exhaustively test its behavior using a stream of randomly generated traffic. For new protocols, developed by private firms for use in private networks, all this knowledge and technology has to be redeveloped from scratch before programmatic implementations are deemed trustworthy. Furthermore, exhaustive testing is impossible or impractical for most implementations, so testing can only give limited assurance.

Many of these protocols are implemented in P4. P4 was introduced in a research paper and quickly commercialized, with an independent consortium overseeing its development. With many competing compiler implementations for many different kinds of programmable data plane, the language has only achieved its current level of stability via standardization by a committee. The published specification is natural language similar to other language standards like the C standard [60], but comparatively short. In part this is due to the simplicity of the language, but it is also due to the incompleteness of the standard which for instance fails to describe the P4 type system in full detail. This is an unsteady foundation for new data plane protocols and implementations.

## 1.2 Certified Systems and Compilation

A certified system is a software or hardware artifact equipped with a specification and a proof in a proof assistant that the artifact meets its specification. Done properly, a certified system makes a strong case for its trustworthiness and reliability to anyone who can read and understand the specification. It reduces the problem of trusting the entire system to the problem of trusting the proof assistant and the system specification, which is sometimes much easier.

If the system is implemented in a high level language like C, a compiler translates the system into the machine code before it is actually executed. This means that the specification proved of the source program only applies to the compiled program if the compiler is correct. Today there are several large verified compilers available: CompCert [70], CakeML [69], and others. While there are properties beyond simple semantics preservation worth proving about a compiler, semantics preservation is the basis for lowering functional correctness guarantees for source programs to functional correctness guarantees for compiled programs.

Compiler correctness has many benefits but its exorbitant costs make it impractical today. The proofs are large and every change to the compiler itself incurs extensive changes in the verification of the compiler. This is a problem for all large proof assistant developments and may be attributed to the absence (or impossibility) of automated proof in the expressive logics of proof assistants. Generally speaking, deciding the correctness of a compiler is impossible. But even proofs of simple lemmas that arise in compiler verification are done manually.

There are alternatives. For complex compiler passes, sometimes it is easier to rely on a certified translation validator. A translation validator is a (semi-)decision procedure for the correctness of a particular run of the compiler. These may be easier to verify than the compiler passes they validate. More importantly, they allow for the

compiler itself to be modified without invalidating old proofs. The drawback is the threat of incompleteness. Translation validation is usually a program equivalence or refinement query, which is undecidable in general, so in existing certified compilers translation validation is reserved for individual passes with decidable validation problems such as register allocation.

In domain-specific languages like P4, translation validation is more attractive in principle because the languages are not Turing complete. However, there are still significant practical obstacles. Historically, most work on translation validation is carried out without foundational proof. These non-foundational validators consume two programs, invoke a SAT/SMT solver repeatedly, and emit a Boolean result. This means that the trusted computing base of compiled code approved by the validator includes the entire validator and the entire SAT/SMT solver it relies on. Removing the SAT/SMT solver and certifying the validator can give much stronger guarantees but only handles simple validation problems and comes at a greater start-up cost than solver-aided alternatives.

## 1.3 Contributions

This thesis gives foundations to the P4 programming language in order to improve the stability of programmable data planes. These contributions address two shortcomings in the state of P4 programming today.

- It is unclear what P4 programs mean. Programs may be misinterpreted by different parties or different articles of language infrastructure and there is no source of truth available to resolve the disagreement.
- It is unclear whether P4 compilers preserve this program meaning, whatever it may be. Today, a program that appears safe may exhibit buggy behavior after it

is compiled.

We first describe what P4 programs mean in in mathematical terms. That is, we give these programs semantics. We begin with a formal definition of the language's syntax in (extended) Backus-Naur Form (BNF). This syntax is then equipped with a type system defined by a collection of inference rules and given operational semantics by a so-called "big step" evaluation relation which relates programs to their possible results. The type system is proved sound with respect to these operational semantics, showing that all P4 programs terminate on all well-typed inputs.

With a clear definition of the semantics of P4 we turn to the problem of P4 compilation. A compiler should preserve program semantics: whatever the compiled program does should correspond to what the source program could do. This basic compiler correctness criterion can be difficult to prove, especially when considering more complex optimizations like those used in P4 compilation to restricted hardware targets. However, because all P4 programs terminate, the language is not Turing complete, so Rice's theorem does not apply to properties of P4 programs. So we develop automatic equivalence checking procedures for validating individual runs of a P4 compiler and apply them to optimizers from the literature. Since the control flow structure of the two P4 sublanguages of parsers and tables are quite different, we use different validation algorithms for each sublanguage.

Our first automated equivalence checker deals with parsers. The verifier, Leapfrog, interprets P4 parsers as finite automata and applies a novel symbolic bisimilarity algorithm to check equivalence. The implementation is mostly in Coq, but relies on a trusted SAT solver to handle certain symbolic computations. The entire system is proved sound in Coq with respect to the automata semantics of P4. P4 parsers are Turing complete, but we impose a productivity condition on parser states that makes our algorithm both sound and complete.

Our second automated equivalence checker addresses match-action table programs. The table language of P4 is free of loops and recursion and can be verified by standard techniques. Here the main complication is that tables are reconfigurable and may be broken up or reallocated by a compiler. So there needs to be a control plane interface that makes sense, complicating the verification problem.

Taken together, these formal semantics and verification tools give stronger foundations to programmable networking and advance the state of the art in foundational compiler verification. In particular, Verified solver-aided translation validators achieve a surprising economy of effort in producing a compact TCB. These results are likely to be replicable in DSLs similar to P4 and may be extended to larger general purpose languages in the future.

CHAPTER 2

# THE P4 PROGRAMMING LANGUAGE

In this chapter we give an overview of P4 as a language with an eye towards defining its semantics. This means we will emphasize reading and understanding the behavior of P4 programs. For a tutorial in writing P4 programs, see the P4 tutorial repository[85].

In broad strokes, P4 is an imperative language with two parts. The first part is a flowchart-like language dedicated to packet parsing. The second part is a loop-free language dedicated to feeding the parsed packet through configurable lookup tables. The datatypes of the language have their quirks, but the language is statically allocated and (outside of the parsing language) terminating. It even features an unusual "copy-in copy-out" calling convention that prevents function arguments from aliasing.

## 2.1 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

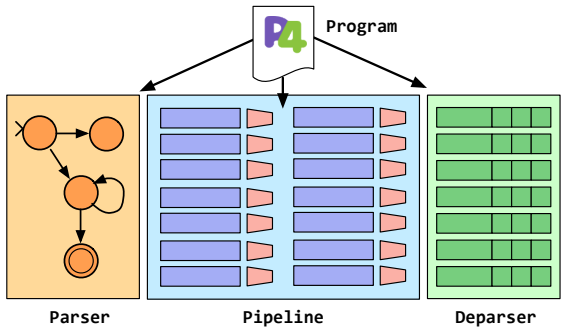



Figure 2.1: Block diagram of a representative three-stage architecture.

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, the block diagram in Figure 2.1 illustrates an 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.

## 2.2 Example Programs

### 2.2.1 Three-Stage Architecture

Figure 2.2 defines the three-stage architecture in P4. 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 programmable components of the architecture: a parser, a pipeline, and a deparser, as well as the `package` that models the device itself.

```
// 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);
```

Figure 2.2: Example: P4 definition of the three-stage architecture.

### 2.2.2 Source Routing with Access Control

The program in Figure 2.3 implements source routing and access control within the three-stage architecture. 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,

```
// 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;
```

Figure 2.3: Example: P4 program that implements source routing with access control in the three-stage architecture.

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.

## 2.3 Language Features

Now that the basic features of architecture definitions and programs have been introduced, we provide a little more detail. Many aspects of P4's semantics are unconventional or complicated, so it can be confusing to digest all at once in a formal semantics. This field guide covers some of the more frequently confusing points now. To check your understanding, refer back to the example code (Figures 2.2 and 2.3) as you read. For all the details, read on to Chapter 3.

### 2.3.1 Data

The type system of P4 includes a subset of data types. Only values of data type may be stored in mutable variables, extracted from packets, and emitted to packets. The

rest of the type system deals with the proper declaration and use of constructs like controls, parsers, tables, packages, and externs. The most basic values are fixed-width integers. For example, the 7-bit integer types come in unsigned (`bit<7>`) and signed (`int<7>`) variants. These basic types may be combined into structure types, which come in struct and header variants. Structs work like struct or record types in other languages. Headers are nullable structs, but in P4 terminology we say that a header is either invalid or valid rather than null or non-null. This feature plays an essential role in parsing, where extract calls only accept headers and mark the header valid once it has been populated.

Invalid headers present a problem for the language semantics because their fields may be read. An invalid read is analogous to memory management mistakes such as dereferencing a null pointer or writing to a deallocated memory region. Addressing this in the static semantics means devising a sound and precise type static analysis that prevents invalid reads from ever occurring, which is not the path taken by P4 but has been the subject of research. Instead the P4 specification introduces a notion of "undefined value" which may be returned from an invalid read. An undefined value is a nondeterministic value of the appropriate type for the invalid read. This nondeterminism means that different reads from the same invalid header might end up returning different values.

### 2.3.2 Controls

Controls are the most common form of procedural abstraction in P4 programming. In the example program there is a single control declaration named `MyPipe`. The declaration begins with a name for the control and a list of parameters. After the parameter list there are curly braces to delimit a sequence of local declarations followed by an `apply` block. The local declarations are similar to private variables and private method

declarations in an object-oriented language, while the `apply` block works like an entry point or invocation method, which again looks object-oriented. The control can be used by instantiating it (just like object-oriented code!) and then calling the instance's `apply` methdod, but in simple programs like the example (Figure 2.3) the control might only be instantiated as part of a package and have its `apply` method invoked only implicitly by the architecture.

The similarity of controls to object-oriented classes can be misleading. Yes, they have local state, can be instantiated, and have at least one method. But there are restrictions, motivated by the unusual hardware P4 was designed to compile to, that make programming with controls feel different. Most significantly, all instantiations must occur at compile time. This means static allocation, not dynamic allocation. There is no subtyping discipline or inheritance mechanism for controls. There is no method overloading. There are no pointers, references, or memory management. All this makes it possible to map controls onto pipelines with bounded and distributed resources.

#### Calling Convention

The P4 calling convention is called "copy-in copy-out" and is dictated by the little `in`, `out`, and `inout` markers on control parameters. Before a function call, a fresh temporary variable is allocated for each parameter marked with a direction. For each in parameter, i.e., each parameter marked either `in` or `inout`, its argument is evaluated and stored in the temporary before the call. For each out parameter, defined analogously to in parameters, the argument must be assignable and following the call the temporary's value is assigned to the argument. By copying everything into temporaries, P4 can provide something like call-by-reference while avoiding the problem of aliasing.

### 2.3.3 Tables

P4 tables are reconfigurable maps which can pattern-match on a key and invoke an action. Each pair of a pattern match and an action is called a rule. A table's list of rules is configured at run time and does not normally appear in the program, although it is possible to specify it at compile time. This reconfigurability is important for integrating P4 programs into existing networks, where reconfigurable rules in routing tables and firewalls are already the most common means of implementing network policy. For instance, consider the ACL or firewell table `acl` from Figure 2.3. A network operator who wants a server A to be able to interact with another server B would have to add a rule to `acl` with pattern (A, B) and action `allow()`.

Table keys are annotated with a match kind, which specifies the kind of pattern that may be used to match that key in the table's rules. The standard match kinds are `exact`, `optional`, and `ternary`, but architectures may provide more. They allow literals as patterns, literals or none/underscore as patterns, and arbitrary bitmasks respectively.

Once a table is declared, it may be invoked by using its `apply()` method. This evaluates the key expression specified in the table declaration, finds the first matching rule, and runs the action specified by the rule.

### 2.3.4 Externs

Externs allow P4 programs to use features of architectures that are not expressible directly in P4. The computational restrictions of P4 mean that even simple P4 programs can end up touching a lot of different externs, depending on the target. In the example, packet parsing and deparsing use externs for I/O.

Like controls, externs are class-like. They even have multiple callable methods instead of just having `apply`. While the only externs in the example are for dealing

with packets, externs are used to declare state that can persist between packets, hash functions, random number generators, and other features.

### 2.3.5 Parsers

Parser declarations look like control declarations but have a collection of states instead of a single apply block. A parser is a state machine, but it may be better to think of it specifically as a flowchart program. In this analogy, the states are the boxes of the flowchart and the `transition` statements are the edges of the flowchart. Parser execution begins in the distinguished `start` state and terminates in the special `accept` or `reject` states, which are not declared in the program.

The most important thing a parser does is invoke the `extract` method of the packet, which is more complex than it may appear at first because it must index into arrays and mark things valid. The signature of this method is `void extract<T>(inout T hdr)` (see Figure 2.2). If the data type `T` has a width of $k$ bits, the `extract` call removes $k$ bits from the front of the packet and stores them in its argument `hdr`. If the argument is a header, it is marked valid. If the argument is a header stack's `.next` field, the bits are appended to the header stack. In practice this involves incrementing an internal offset associated with the header stack. If the argument is a field of a header union, that field is marked valid and all other fields of the union are marked invalid.

CHAPTER 3

# PETR4

## 3.1 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 [103]. 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) [67], Anteater [75], NetKAT [3], Batfish [40], Minesweeper [11], ARC [43], 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 [35], Cisco [23], and Microsoft [14, 72] 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 [17, 104] 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 [53], but the language is flexible enough to implement or specify completely new forwarding behavior—e.g., in-band telemetry [54] or in-network computing [62, 61] 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 chapter 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

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

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 chapter makes the following contributions:

- We develop a clean-slate, definitional interpreter for the P4 language ($p$ 3.4).
- We define a calculus that models the semantics of a core fragment of P4 in terms of standard typing and operational semantics judgments. ($p$ 3.3).
- We prove type soundness and termination ($p$ 3.3) for our calculus.
- We develop an extension to the language ($p$ 3.6) 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 ($p$ 3.5).

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.

## 3.2 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.

**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 2.3 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 [73, 101] 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 [93]. 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 [98] 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 2.3 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 2.3 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.

## 3.3 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. 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."

### 3.3.1 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 2.3:

```
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.

#### 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

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

Figure 3.1: Core P4 types and directions.

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 $\ell$.

**Types (Figure 3.1).**

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:

```
header { copyFlag : bit⟨1⟩
         optClass : bit⟨2⟩
         option : bit⟨5⟩
         optionLength : bit⟨8⟩ }
```

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

| | | | | | |
|---|---|---|---|---|---|
| *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 | | | |

Figure 3.2: 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.

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.

```
const int w := 8; bit⟨w⟩ x := 1; bit⟨8⟩ y := x;
```

The type bit⟨w⟩ is not syntactically equal to bit⟨8⟩, 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 [78]. Repeated declarations extend the open enumeration with new members without replacing the old members or shadowing the existing type.

**Expressions (Figure 3.2)**

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:

```
const bit⟨48⟩ bits := ...;
const int n := 6;
bit⟨8⟩ nth_byte := bits[n * 8 − 1:(n − 1) * 8]
```

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.

**Statements (Figure 3.2)**

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:

```
function {} f(in h : hdr_t) {if (!isValid(h.ip)) {exit} else {...}}
function {} g(in h : hdr_t) {f(h); f(h)}
```

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

| | | | |
|---|---|---|---|
| *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 |

Figure 3.3: Core P4 declarations and programs.

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.

**Declarations and Programs (Figure 3.3)**

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.

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" $\mathscr{C}$ 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.

**L-Values (Figure 3.2)**

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

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

Figure 3.4: 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.

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 3.3.2).

**Values and Signals (Figure 3.4)**

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, remove, or add a tag. Header and stack values include their field and element types to facilitate our treatment of undefined reads (see Section 3.3.3).

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,

| | | | |
|---|---|---|---|
| $\Gamma$ | ::= | $\Gamma,\ x_1 : \tau_1$ | 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 → Value | constant context |
| $\sigma$ | : | Loc → Value | store |
| $\epsilon$ | : | Var → Loc | environment |
| $\Xi$ | : | Loc → Type | store typing context |
| $\mathscr{C}$ | : | Loc × Value × $\overline{\text{PartialActRef}}$ → ActRef | control plane |

Figure 3.5: 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.

hash functions.

bit⟨16⟩ hash_crc16⟨T⟩(in data : 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.

**Typing and Evaluation Contexts (Figure 3.5)**

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

| | | | |
|---|---|---|---|
| $\Sigma, \Gamma, \Delta \vdash exp : \tau$ 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 | | |

Figure 3.6: Selected judgment signatures from the static semantics.

values.

### 3.3.2 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 3.4 for details).

Typing judgments are given in Figure 3.6. 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 $c+1$.

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

**Expression Typing (Figure 3.6)**

The expression typing judgment is defined in Figure 3.7. 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 $d \neq \mathsf{in}$ 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 $\mathscr{T}$. 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.

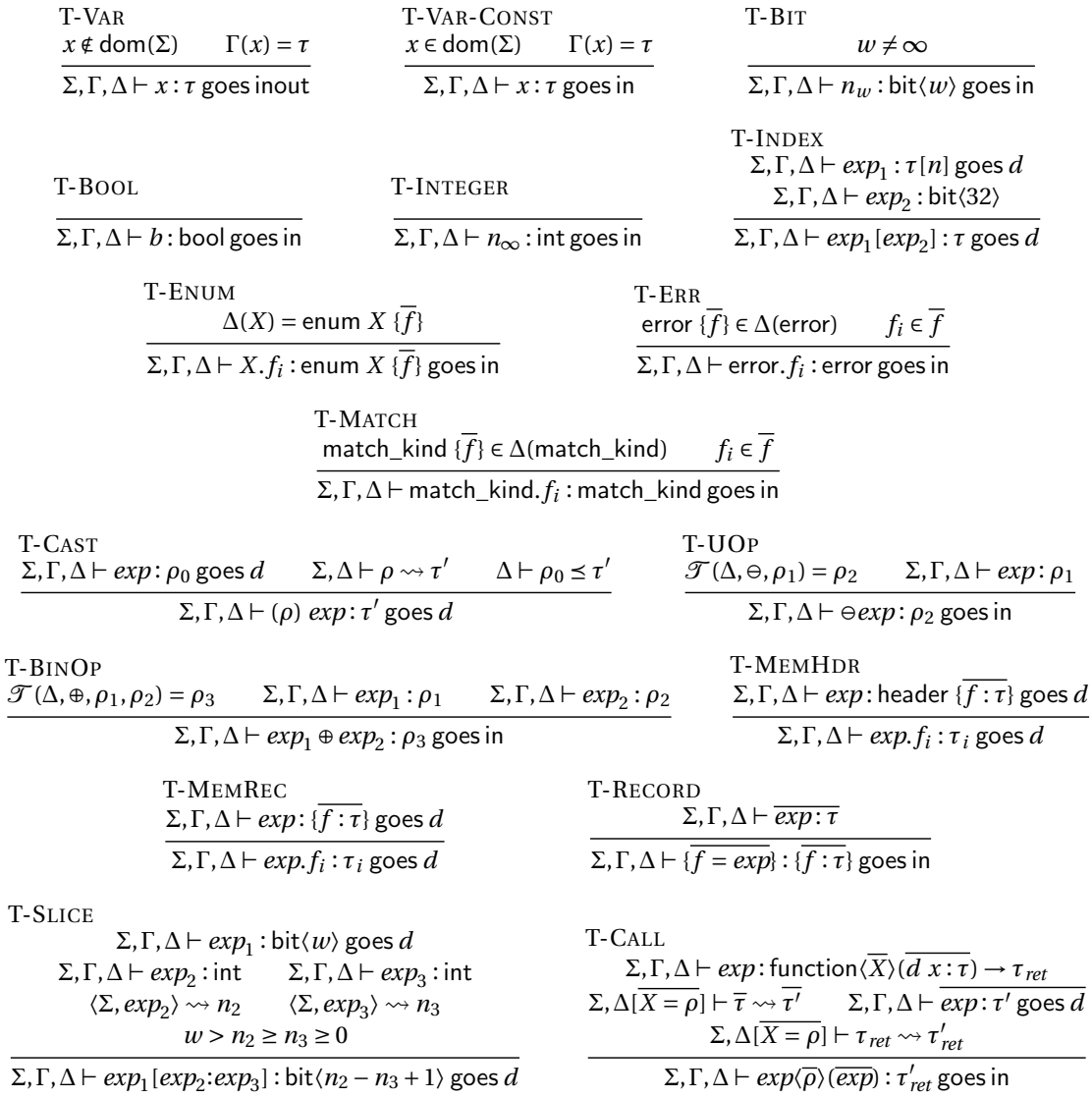



Figure 3.7: Expression typing rules.

**Statement Typing (Figure 3.8)**

The typing rules for statements, defined in Figure 3.8, 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

$$\textsc{TS-Empty}\ \frac{}{\Sigma,\Gamma,\Delta \vdash \{\} \dashv \Sigma,\Gamma} \qquad \textsc{TS-Exit}\ \frac{}{\Sigma,\Gamma,\Delta \vdash \mathsf{exit} \dashv \Sigma,\Gamma}$$

$$\textsc{TS-Block}\ \frac{\Sigma,\Gamma,\Delta \vdash stmt \dashv \Sigma_1,\Gamma_1 \qquad \Sigma_1,\Gamma_1,\Delta \vdash \{\overline{stmt}\} \dashv \Sigma_2,\Gamma_2}{\Sigma,\Gamma,\Delta \vdash \{stmt;\overline{stmt}\} \dashv \Sigma,\Gamma}$$

$$\textsc{TS-Decl}\ \frac{\Sigma_0,\Gamma_0,\Delta_0 \vdash var\_decl \dashv \Sigma_1,\Gamma_1,\Delta_1}{\Sigma_0,\Gamma_0,\Delta_0 \vdash var\_decl \dashv \Sigma_1,\Gamma_1}$$

$$\textsc{TS-Assign}\ \frac{\Sigma,\Gamma,\Delta \vdash exp_1 : \tau\ \mathsf{goes\ inout} \qquad \Sigma,\Gamma,\Delta \vdash exp_2 : \tau}{\Sigma,\Gamma,\Delta \vdash exp_1 := exp_2 \dashv \Sigma,\Gamma}$$

$$\textsc{TS-Ret}\ \frac{\Sigma,\Gamma,\Delta \vdash exp : \tau \qquad \Sigma,\Delta \vdash \Gamma(\mathsf{return}) \rightsquigarrow \tau}{\Sigma,\Gamma,\Delta \vdash \mathsf{return}\ exp \dashv \Sigma,\Gamma}$$

$$\textsc{TS-If}\ \frac{\Sigma,\Gamma,\Delta \vdash exp : \mathsf{bool} \qquad \Sigma,\Gamma,\Delta \vdash stmt_1 \dashv \Sigma_1,\Gamma_1 \qquad \Sigma,\Gamma,\Delta \vdash stmt_2 \dashv \Sigma_2,\Gamma_2}{\Sigma,\Gamma,\Delta \vdash \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2 \dashv \Sigma,\Gamma}$$

$$\textsc{TS-TblCall}\ \frac{\Sigma,\Gamma,\Delta \vdash exp : \mathsf{table}}{\Sigma,\Gamma,\Delta \vdash exp() \vdash \Sigma,\Gamma} \qquad \textsc{TS-Call}\ \frac{\Sigma,\Gamma,\Delta \vdash exp\langle\overline{\rho}\rangle(\overline{exp}) : \tau}{\Sigma,\Gamma,\Delta \vdash exp\langle\overline{\rho}\rangle(\overline{exp}) \dashv \Sigma,\Gamma}$$

Figure 3.8: Statement typing rules.

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 Figure 3.10) insert a type for return before typechecking the bodies of functions and controls.

### Variable Declaration Rules (Figure 3.9)

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.

### Object Declaration Rules (Figure 3.10)

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

$$\textsc{Type-Const}\ \frac{\Sigma,\Delta \vdash \tau \rightsquigarrow \tau' \qquad \Sigma,\Gamma,\Delta \vdash exp : \tau' \qquad \langle \Sigma, exp\rangle \rightsquigarrow v}{\Sigma,\Gamma,\Delta \vdash \mathsf{const}\ \tau\ x := exp \dashv \Sigma[x = v],\Gamma[x : \tau'],\Delta} \qquad \textsc{Type-Var}\ \frac{\Sigma,\Delta \vdash \tau \rightsquigarrow \tau'}{\Sigma,\Gamma,\Delta \vdash \tau\ x \dashv \Sigma,\Gamma[x : \tau'],\Delta}$$

$$\textsc{Type-VarInit}\ \frac{\Sigma,\Delta \vdash \tau \rightsquigarrow \tau' \qquad \Sigma,\Gamma,\Delta \vdash exp : \tau'}{\Sigma,\Gamma,\Delta \vdash \tau\ x := exp \dashv \Sigma,\Gamma[x : \tau'],\Delta} \qquad \textsc{Type-Inst}\ \frac{\Sigma,\Gamma,\Delta \vdash C : \mathsf{ctor}(\overline{x : \tau}) \to \tau_{inst} \qquad \Sigma,\Gamma,\Delta \vdash \overline{exp : \tau}}{\Sigma,\Gamma,\Delta \vdash X(\overline{exp})\ x \dashv \Sigma,\Gamma[x : \tau_{inst}],\Delta}$$

Figure 3.9: Variable declaration typing rules.

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

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

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

Figure 3.10: Object declaration typing rules.

return statements within the body of the declaration. As discussed earlier, the type {} is an empty record type representing P4's `void` return type.

### 3.3.3 Dynamic Semantics

The dynamic semantics for Core P4 is defined in a big-step style. Figure 3.11 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

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

Figure 3.11: 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.

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 $\mathsf{havoc}(\tau)$ to indicate an operation producing an arbitrary value of type $\tau$. Match-action evaluation uses the control plane $\mathscr{C}$ 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.

### Copy-In Copy-Out Rules (Figure 3.12)

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

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

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

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

Figure 3.12: Copy-in and copy-out operations. We define them for single arguments and they are lifted to lists of arguments in the obvious way.

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).

$$\mathsf{function}\ \{\}\ f(\mathsf{inout}\ \mathsf{bit}\langle 8\rangle\ src, \mathsf{inout}\ \mathsf{bit}\langle 8\rangle\ dst)\ \{dst := src + 1;\ src := 0\}$$

$$x := 1;$$

$$f(x, x);$$

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).

**Expression Evaluation (Figures 3.13 and 3.14).**

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 $\mathscr{T}$ and an evaluation function $\mathscr{E}$. For casts, this means there is agreement between a casting check $\Delta \vdash \tau \preceq \tau'$ and a casting function $\mathsf{cast}(\Sigma, val, \tau)$.

$$\text{E-Int}\ \frac{}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, n_w \rangle \Downarrow \langle \sigma, n_w \rangle} \qquad \text{E-Bool}\ \frac{}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, b \rangle \Downarrow \langle \sigma, b \rangle} \qquad \text{E-TypMem}\ \frac{}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, X.f \rangle \Downarrow \langle \sigma, X.f \rangle}$$

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

$$\text{E-Uop}\ \frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma', val \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, \ominus exp \rangle \Downarrow \langle \sigma', \mathscr{E}(\ominus, val) \rangle} \qquad \text{E-BinOp}\ \frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp_1 \rangle \Downarrow \langle \sigma_1, val_1 \rangle \qquad \langle \mathscr{C}, \Delta, \sigma_1, \epsilon, exp_2 \rangle \Downarrow \langle \sigma_2, val_2 \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp_1 \oplus exp_2 \rangle \Downarrow \langle \sigma_2, \mathscr{E}(\oplus, val_1, val_2) \rangle}$$

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

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

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

Figure 3.13: Semantics for expressions I.

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 [109]. 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 $\mathscr{N}$.

**Variable Declaration Evaluation (Figure 3.15)**

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

E-HDRMEM

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

E-HDRMEMUNDEF

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

E-CALL-DECLEXIT

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

E-CALL-STMTEXIT

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

E-CALLN

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

E-CALL

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

E-SLICE

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

Figure 3.14: Semantics for expressions II.

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

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

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

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

Figure 3.15: Semantics for variable declarations.

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

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

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

Figure 3.16: Semantics for object declarations.

the constructor arguments in the store and closure environment.

### Object Declaration Evaluation (Figure 3.16)

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.

E-EMPTY

$$\frac{}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, \{\} \rangle \Downarrow \langle \sigma, \epsilon, \mathsf{cont} \rangle}$$

E-EXIT

$$\frac{}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, \mathsf{exit} \rangle \Downarrow \langle \sigma, \epsilon, \mathsf{exit} \rangle}$$

E-IFT

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

E-IFF

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

E-BLOCK

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

E-RETURN

$$\frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, val \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, \mathsf{return}\ exp \rangle \Downarrow \langle \sigma_1, \epsilon, \mathsf{return}\ val \rangle}$$

E-ASSIGN

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

E-CALL-TABLE

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

E-VARDECL

$$\frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, var\_decl \rangle \Downarrow \langle \Delta', \sigma', \epsilon', \mathsf{cont} \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, var\_decl \rangle \Downarrow \langle \sigma', \epsilon', \mathsf{cont} \rangle}$$

E-CALL

$$\frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp\langle \overline{\rho} \rangle (\overline{exp}) \rangle \Downarrow \langle \sigma', sig \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp\langle \overline{\rho} \rangle (\overline{exp}) \rangle \Downarrow \langle \sigma', \epsilon, sig \rangle}$$

Figure 3.17: Semantics for statements.

**Statement Evaluation (Figure 3.17).**

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

### 3.3.4 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.

### 3.3.5 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 3.3.1.** *Let* $\langle \mathscr{C}, \Delta, \sigma, \epsilon, stmt \rangle$ *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. $\Sigma, \Gamma, \Delta \vdash stmt \dashv \Sigma', \Gamma'$.

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

1. $\langle \mathscr{C}, \Delta, \sigma, \epsilon, stmt \rangle \Downarrow \langle \sigma', \epsilon', sig \rangle$,
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.

## 3.4 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 3.18 (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 [63]: 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*

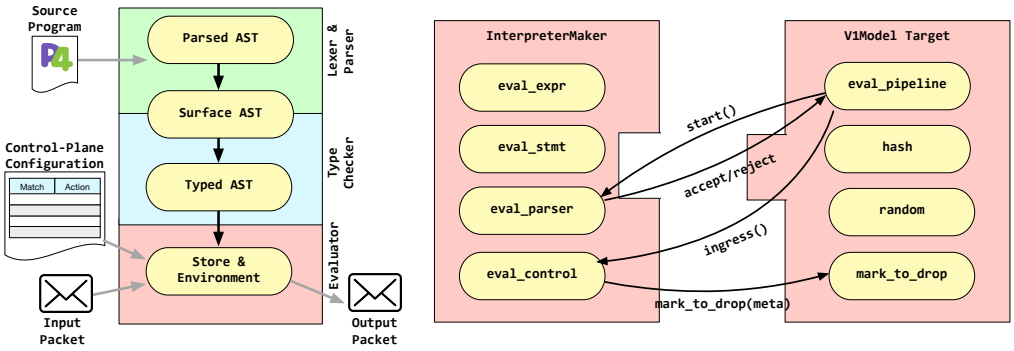
> *argument, when the value of [sic] returned in that argument is ignored by subsequent computations. When used in generic functions or methods, the compiler may reject the program if it is unable to infer a type for the don't care argument.*

However, aside from a brief mention of the Hindley-Milner inference algorithm [28], 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 [41]. What is more surprising is that Hindley-Milner is unnecessary for $\text{P4}_{16}$. 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 [88], 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.

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 PETR4 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 specifica-




(a) (b)

Figure 3.18: PETR4 implementation: (a) interpreter data flow; (b) architecture support via plug-ins.

tion 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 PETR4 interpreter implements a big-step evaluator, following the same basic approach as the Core P4 evaluation relation (Section 3.3). However, whereas Core P4 uses nondeterminism to overapproximate possible target-specific behaviors, the PETR4 interpreter uses a "plugin" approach. The interpreter is an OCaml functor with the following signature:

```
functor (T : Target) → 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:

```
extern : env → state → type list → (value × type) list → env × state × 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.

```
eval_pipeline : ctrl → env → state → buf → apply → state × env × 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, PETR4 does not currently support multicast, but adding it would be a relatively straightforward extension.

Figure 3.18 (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:

```
eval_program : ctrl → env → state → buf → int → prog → state × (buf × int) 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

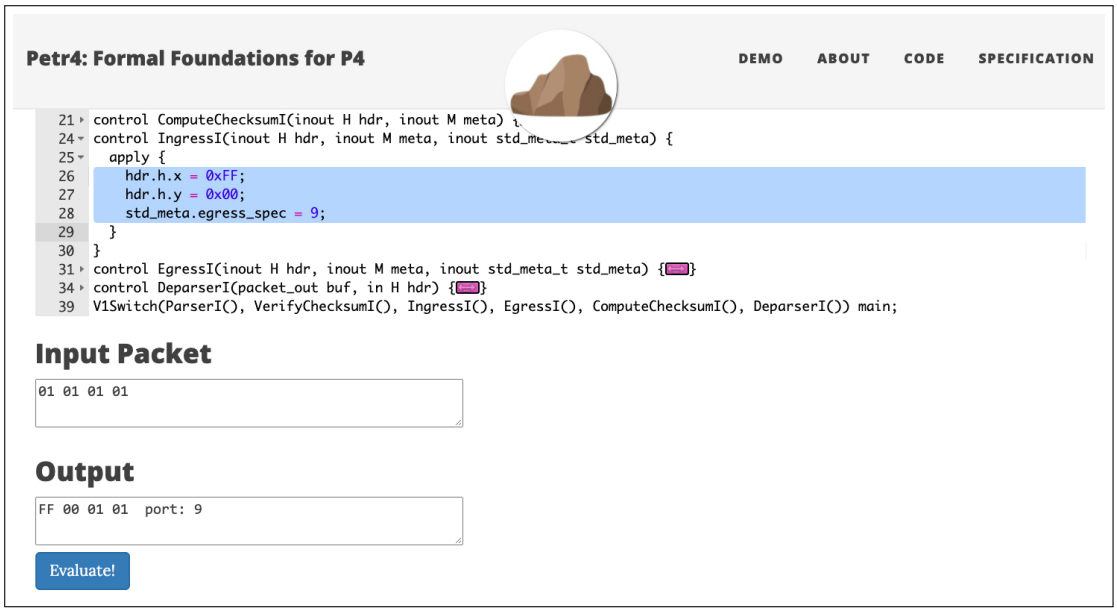



Figure 3.19: Petr4 interpreter running in a web browser. The interpreter is online at cornell-netlab.github.io/petr4.

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 3.19. 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.

### 3.4.1 Limitations

PETR4 implements the vast majority of features discussed in the $P4_{16}$ specification. However, our current prototype does have some important limitations. PETR4 imple-

ments 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).

## 3.5 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 3.4.1 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

| 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 | ✗ |

Figure 3.20: Selection from P4C's STF test suite.

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 3.20. 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 3.21.

| 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 | ✓ |

Figure 3.21: Selection from PETR4's custom STF test suite.

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 3.22 and Figure 3.23 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

| 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. |
| | (b) The parser incorrectly resolved names with a "dot" using the local context instead of the top-level context. |
| | (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. |
| | (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. |
| | (f) The front-end's constant folding transformed a program that was not well-typed into one that was. |
| | (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`. |
| | (i) The compiler did not clearly enforce that values of type `int` should all be compile-time known values. |
| | (j) An STF test had two lines uncommented that were supposed to be commented. |
| | (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`. |
| | (l) The compiler did not stop compiling after encountering an error. |

Figure 3.22: P4C Issues

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.

| 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. |
| | (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. |
| | (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. |
| | (e) The spec is too restrictive because it only permits division and modulo between positive `int` values. |
| | (f) The spec does not explicitly allow header unions in header stacks. |
| | (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. |

Figure 3.23: $P4_{16}$ Specification Issues

**Typechecker.** 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. 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.

**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.

**Typing Mistakes.** 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.

## 3.6 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.

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

**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).

T-UNION

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

T-SWITCH

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

**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:

E-UNION

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

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.

E-UNIONSWITCH

$$\frac{\langle \mathscr{C}, \Delta, \sigma, \epsilon, exp \rangle \Downarrow \langle \sigma_1, X\{f_i, val_i\} \rangle \quad \mathsf{match\_union\_case}\ (\overline{lbl\colon \{\overline{stmt}\}}, f_i) = (f_i, k) \qquad \ell\ \mathsf{fresh} \quad \sigma_2 = \sigma_1[\ell \mapsto val_i] \qquad \langle \mathscr{C}, \Delta, \sigma_2, \epsilon[f_i \mapsto \ell], \{\overline{stmt}_k\} \rangle \Downarrow \langle \sigma_3, \epsilon', sig \rangle}{\langle \mathscr{C}, \Delta, \sigma, \epsilon, \mathsf{switch}\ (exp)\ \{\overline{lbl\colon \{\overline{stmt}\}}\} \rangle \Downarrow \langle \sigma_3, \epsilon, sig \rangle}$$

**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:

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

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.

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

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 $f_i$.

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

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

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

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

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

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

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

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 [36] 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 3.6.1.** *If* $\langle \mathscr{C}, \Delta, \sigma, \epsilon, stmt\rangle \Downarrow \langle \sigma', \epsilon', sig\rangle$, *then* $\langle \mathscr{C}, \Delta, [\![\sigma]\!], \epsilon, [\![stmt]\!]\rangle \Downarrow \langle \sigma_t, \epsilon_t, sig\rangle$ *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* $\sigma_2(\ell_2) = val$.

## 3.7 Conclusion and Future Work

This chapter 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.

We are grateful to the POPL'21 reviewers for their feedback and suggestions for improving this chapter. 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 Pedro Amorim, Griffin Berlstein, Cal Gunnarsson, Tobias Kappé,

Rolph Recto, John Sarracino, Steffen Smolka, and Doug Woos for their 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.

CHAPTER 4

# LEAPFROG: CERTIFIED EQUIVALENCE FOR PROTOCOL PARSERS

## 4.1 Introduction

Devices like routers, firewalls and network interface cards as well as operating system kernels occupy a critical role in modern communications infrastructure. Each of these implements parsing for a cornucopia of networking protocols in its *protocol parser*. The parser is the network's first line of defense, responsible for organizing and filtering unstructured and often untrusted data as it arrives from the outside world. Due to their crucial role, bugs in parsers are a significant source of crashes, vulnerabilities, and other faults [94].

**Example Router Bug.** Consider the following bug, which was present in a commercial router developed by a leading equipment vendor several years ago. Internally, the router was organized around a high-throughput pipeline, which most packets traversed in a single pass. However some packets had to be *recirculated*, meaning they took additional passes through the pipeline before being sent back out on the wire. The router used an internal state variable to decide whether a packet should be recirculated. Usually this state variable was initialized by vendor-supplied code. But, as was discovered by a customer, it could also be erroneously initialized from data in non-standard, malformed packets. Hence, crafted packets could bypass the vendor-supplied initialization code, resulting in an infinite recirculation loop—a denial-of-service (DoS) attack on the router and its peers. In the presence of broadcast traffic, such a "packet storm" would monopolize the router's resources, rendering it unusable until it was rebooted.

An easy way to avoid this bug would be to modify the router's parser to filter away

malformed packets, while still accepting valid packets. However, to have full confidence in the new parser, one would need to prove that it is *equivalent* to the original, modulo malformed packets. Although parsers tend to be simple, this would likely be a challenging verification task—it requires reasoning about a *relational* property across two distinct programs.

**Parser Equivalence Checking.** This chapter studies relational verification of protocol parsers, focusing specifically on equivalence of parsers expressed in terms of state machines. Semantic equivalence [22] is a fundamental problem that underpins a wide range of practical verification tasks including translation-validation [81], superoptimization [76], and program synthesis [50]. As we will see in Section 4.7, the algorithm that we develop for computing equivalence can also be straightforwardly extended to other relational verification challenges, including one inspired by the router bug above (c.f. our external filtering case study in Section 4.7.1).

There are several technical challenges related to mechanically and formally proving protocol parser equivalence. First, we need a computational model for parsers that is expressive enough to handle practical parsers, but also sufficiently restricted to enable tractable formal verification. Second, we need efficient reasoning techniques based on symbolic representations and domain-specific insights to handle the enormous state space of real-world parsers. Third, we need effective automation and tool support so programmers can avoid manually crafting sprawling proofs of equivalence.

**Certified Tooling.** The foundational guarantees offered by proof assistants are highly desirable in error-prone domains. However, achieving these guarantees is notoriously hard, as proof assistants need an experienced engineer's guidance to prove all but the simplest goals. One way to scale verification is to break systems into smaller components that compose along shared specifications [5]. This allows individual

verification tasks to be solved in isolation, without compromising top-level guarantees. Our hope is that push-button verifiers can reduce total proof burden by certifying some components automatically.

Consider the task of proving that a realistic parser meets a functional specification, in the style of VST [6] proofs which relate C programs to functional specifications. The parser might be hand-optimized for performance reasons like the vectorized parser in Figure 4.1, which makes it difficult to reason about directly. With an automated equivalence checker, we could justify replacing it with the simpler and easier to verify reference implementation. Other problems like translation validation [107, 66] or proof-producing synthesis [111] would similarly benefit from certified tooling.

**Our Contribution: Leapfrog.** We present Leapfrog,[1] a new framework that addresses these challenges. It provides an expressive automata model for parsers, with syntax inspired by P4 [16, 104], a networking DSL. The model captures common programming idioms and offers a domain-specific interface for packet parsing. We demonstrate its applicability by encoding parsers for real-world protocols like IPv4 and MPLS.

To establish the equivalence of Leapfrog parsers, we extend classical techniques based on bisimulations to work with symbolic representations of the state space. We also develop a novel up-to technique based on "leaps" that dramatically reduces the cardinality of the constructed relation.

We implement Leapfrog as a Coq library. This allows us to mechanize our metatheory and produce certificates. Our algorithm, which runs inside the Coq prover, produces reusable Coq theorems of parser equivalence. At a technical level, our Coq development combines classical techniques based on predicate transformers, domain-specific optimizations, and a plugin to interface with SMT solvers, to facilitate effective automation. We apply Leapfrog to several benchmarks and find that it is able to scale

[1] `https://github.com/verified-network-toolchain/leapfrog`

up to handle realistic protocols.

The contributions of this chapter are as follows:

- We develop a parser model based on automata extended with domain-specific features (Section 4.3).
- We design efficient algorithms for establishing the equivalence of parsers based on symbolic and up-to techniques (Sections 4.4 and 4.5).
- We realize our approach in a Coq-based framework for automatically constructing equivalence proofs (Section 4.6). Crucially, our design integrates off-the-shelf SMT solvers into a verification loop within Coq.
- We explore Leapfrog's expressiveness and scalability (Section 4.7), finding that it can handle common protocol verification challenges and can perform translation validation for an existing parser compiler.

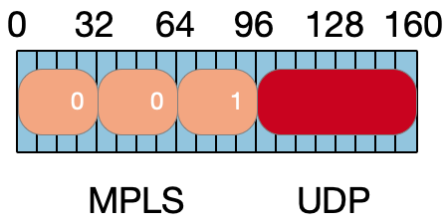
Overall, we believe Leapfrog represents a promising step toward the vision of certified proofs for protocol parsers.

## 4.2 Overview

We now give a high-level overview of our automata model and equivalence checking framework through an illustrative example. Suppose we would like to parse packets with MPLS [108] and UDP [90] headers. An MPLS header is a sequence of 32-bit *labels*. Rather than prefix this sequence with its length, the MPLS format marks the end of the sequence with a label whose 24th bit is 1. This is analogous to the role of the null terminator in C strings. In our example, the MPLS header is followed by an 8-byte UDP header.

We can parse this format in our *P4 automata* model of parsers, which is based on a subset of P4. A P4 automaton is a state machine that parses a packet bitstring

```
q1 {
  extract(mpls, 32);
  select(mpls[23:23]) {
    0 ⇒ q1
    1 ⇒ q2
  }
}
q2 {
  extract(udp, 64);
  goto accept
}
```




```
q3 {
  extract(old, 32);
  extract(new, 32);
  select(old[23:23], new[23:23]) {
    (0, 0) ⇒ q3
    (0, 1) ⇒ q4
    (1, _) ⇒ q5
  }
}
q4 {
  extract(udp, 64);
  goto accept
}
q5 {
  extract(tmp, 32);
  udp := new ++ tmp;
  goto accept
}
```

Figure 4.1: Reference (q1, q2) and vectorized (q3, q4, q5) parsers for MPLS and UDP headers (depicted inset).

into a collection of *headers* stored in global variables, ultimately either accepting or rejecting the packet. Each state in a P4 automaton contains a program that may assign to variables and extract some number of bits from the front of the packet into a variable. For instance, an `extract(h, 64)` operation removes 64 bits from the front of the packet and stores them into the header `h`. Next, the machine transitions to a new state by branching on the contents of its header variables. This is accomplished with either an unconditional transition of the form `goto state` or a conditional transition of the form `select(e) { pat => state }`. A `select` evaluates `e` and transitions to the state associated with the first matching pattern in `pat`.

We give two P4 automata for our format in Figure 4.1. The first automaton is a reference parser with a state `q1` that parses MPLS and another state `q2` that parses UDP. The second automaton has been *vectorized* to parse two MPLS labels at a time in state `q3`. When the second label is the bottom of the stack, the vectorized parser goes on to handle UDP normally (`q4`). If the first label is the bottom of the stack, however,

the vectorized parser marshals the 32 bits of the ill-fated second label into UDP, along with the remaining 32 bits of UDP header data remaining in the packet (`q5`).

The two parsers in Figure 4.1 each use different names for the header (e.g., `mpls` vs. `old/new`). Also, those headers are overwritten multiple times—a real P4 parser would use an array-like data structure called a *header stack* to store the labels [104, Section 8.17]. Our language does not support header stacks directly, although they can be emulated. Here, we omit this detail for simplicity, and we focus on proving that the parsers accept the same sets of packets. Leapfrog can also be used to prove relational properties involving the header values—see Section 4.7 for details.

**Tractable Equivalence Checking.** Our equivalence algorithm is inspired by Moore's algorithm [55] applied to the domain of P4 automata. It is a worklist-style algorithm that begins with a coarse approximation of language equivalence and iteratively refines it by analyzing the joint state space of the two automata. P4 automata have finitely many states and their header variables are fixed-size bitvectors, so their configuration spaces are finite. However, because of the large bitvectors encoded in their stores, P4 automata may have an intractably large configuration space. For instance, the automata in Figure 4.1 have a joint configuration space on the order of $2^{128} \approx 10^{38}$ states! So, naive bisimulation-based approaches will never be tractable for realistic automata. We address this challenge by representing large relations *symbolically*, rather than keeping track of concrete sets of configuration pairs.

Furthermore, we prune the configuration space from the start using a simple reachability analysis. This lets us avoid spurious search steps through unreachable configurations.

In the automata-theoretic semantics of the reference MPLS parser, the `extract(udp,64)` call performs 64 steps that read one bit of the packet into the buffer. The 64th step empties the buffer into the `udp` header variable, and transitions to the accept state. This

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $h$ | $\in$ | $H$ | header names | $q$ | $\in$ | $Q \cup \{\mathsf{accept}, \mathsf{reject}\}$ | state names |
| $n$ | $\in$ | $\mathbb{N}$ | natural numbers | $c$ | ::= | $\overline{pat} \Rightarrow q$ | select case |
| $bv$ | $\in$ | $\{0,1\}^*$ | bitvector | $tz$ | ::= | $\texttt{goto}(q)$ | direct |
| $e$ | ::= | $h$ | headers | | \| | $\texttt{select}(\overline{e})\{\overline{c}\}$ | select |
| | \| | $bv$ | bitvectors | $op$ | ::= | $\texttt{extract}(h)$ | extract |
| | \| | $e[n_1 : n_2]$ | bitslices | | \| | $h := e$ | assign |
| | \| | $e_1 \mathbin{+\!\!+} e_2$ | concatenation | | \| | $op_1; op_2$ | sequence |
| $pat$ | ::= | $bv$ | exact match | $st$ | ::= | $q\,\{op; tz\}$ | states ($q \in Q$) |
| | \| | _ | wildcard | $aut$ | ::= | $\overline{st}$ | P4 automaton |

Figure 4.2: Internal syntax for P4 automata.

bit-by-bit approach is needed to relate parsers that read the packet in differently-sized chunks—as is the case for states q1 and q3 in Figure 4.1. However, a naive search for a bisimulation that treats each step separately would have a huge symbolic state and too many SMT queries. To counteract this, we introduce *bisimulations with leaps*, which keeps the symbolic state compact and avoids redundant SMT queries, processing multiple consecutive steps in each iteration. Together, these optimizations make it feasible to compute bisimulations for realistic parsers.

**Certifying Parser Equivalence.** To make Leapfrog usable in larger developments, we need it to produce reusable proof certificates that can be checked by the Coq kernel. Obtaining full certificates of equivalence from a push-button tool is a significant engineering challenge. Rather than write a solver in Coq for our verification conditions, which sit roughly in the first-order theory of bitvectors, we chose to use external SMT solvers. This had engineering and performance benefits, but getting Coq and external solvers to work together still posed several challenges. First, existing interfaces between Coq and SMT solvers did not meet our needs. We tried using existing plugins for proving Coq theorems with external solvers [27, 7], but found that they scaled poorly or lacked support for key logical operators. To address this, we developed a first-order theory of bitvectors in Coq, as well as a plugin that pretty-prints this logic in SMT-LIB format [10] and discharges the query using an off-the-shelf solver. We did not

implement proof reconstruction, which converts the SMT solver refutation into a Coq proof term. Consequently the solver output and our pretty-printer must be trusted, although this restriction could be lifted in future work.

Second, in order to target this low-level logic and improve the scalability of our tool, we developed and verified a chain of compilation steps that go from the high-level logic used in our algorithm to the low-level logic sent to solvers. This process compiles away features like finite maps, but also performs algebraic simplifications and other rewrites to keep the size and complexity of our SMT queries under control.

Third, in order to communicate with SMT solvers, our algorithm needed to perform I/O, which Coq programs are generally not allowed to do. Coq tactics, however, are allowed to perform I/O, because the proof they produce is still checked by the Coq kernel. We rephrased our algorithm as a proof search problem (c.f. Figure 4.4) and developed a custom Coq tactic as an escape hatch, allowing the algorithm to consult an SMT solver while still producing a Coq certificate (modulo the soundness of the solver and our plugin).

## 4.3 Parser Model

We now describe *P4 automata* (P4As), an automata-theoretic model close to P4's parsing language [24]. A P4A is a state machine that (1) decides whether to accept a packet and (2) builds a data structure (the *store*) using the packet data. The store consists of bitvectors called *headers*,[2] and is a representation of the (partially) parsed packet used within the parser, but also in later processing phases. If a P4A consumes data in each state, it terminates on finite packets.[3]

For example, state `q1` of the reference MPLS/UDP parser in Figure 4.1 extracts 32

[2]Initial header values are undefined in P4 [24, Sections 6.7 and 8.22]; our semantics considers them part of the packet.

[3]Such a restriction is allowed by P4 specification [24, Sec. 12.10].

bits into the `mpls` header, before looping back to `q1` or transitioning to `q2` to parse a UDP header.

Concretely, a P4A is composed of states, each of which acts in two steps: first, it runs its internal program, which consumes some bits from the packet and updates the headers; then, it decides (based on the store) whether to accept (resp. reject) the packet by transitioning to the accept (resp. reject) state, or continue processing the remainder elsewhere. This second step defines the state's transitions.

### 4.3.1 Syntax

The syntax for P4As is best understood by example. Consider the two programs in Figure 4.1. Together, these contain five states, named `q1` through `q5`. Each state contains code, consisting of assignments and `extract` statements, and ending in a `select` or `goto` statement that defines the outgoing transitions. Headers function as variables whose scope and lifetime is shared between states. For instance, in `q5`, the vectorized parser extracts bits into `tmp`, and then stores the contents of `new` and `tmp` in `udp`, before accepting.

We formally define the syntax in Figure 4.2, parameterized over a finite set of states $Q$, and a finite set of *header names* $H$. Each header $h \in H$ has an associated *size* $\mathsf{sz}(h) \in \mathbb{N}^+$. We refrain from specifying these parameters explicitly, as they can be inferred from the program text. For instance, in the P4A on the left in Figure 4.1, $Q = \{\texttt{q1}, \texttt{q2}\}$, and $H = \{\texttt{mpls}, \texttt{udp}\}$, while header sizes are $\mathsf{sz}(\texttt{mpls}) = 32$ and $\mathsf{sz}(\texttt{udp}) = 64$.

P4As associate with each state $q \in Q$ an operation block $op(q)$ and transition block $tz(q)$. Crucially, we require that least one call to `extract` appears in the operations of each state. This guarantees that each state makes some progress on the packet, which ensures termination of both the parsing process and our equivalence checking algorithm.

### 4.3.2 Semantics

To provide a semantics for P4As, we first assign a semantics to the operation and transition code associated with each state. We fix a P4A *aut* with states $Q$, headers $H$ and sizes sz. We write $|bv|$ for the length of $bv \in \{0,1\}^*$. We define $S$ as the finite set of functions $s : H \to \{0,1\}^*$ where $|s(h)| = \mathsf{sz}(h)$.

We first give a semantics to expressions.

**Definition 4.3.1** (Expression Semantics)**.** Let $w, x \in \{0,1\}^*$. We write $wx$ for their concatenation. If $n_1, n_2 \in \mathbb{N}$, we write $w[n_1 : n_2]$ for the zero-indexed substring starting at position $\min(n_1, |w|-1)$ and ending at $\min(n_2, |w|-1)$, inclusive.

Given an expression $e$, we inductively define its semantics in the form of a function $[\![e]\!]_{\mathscr{E}} : S \to \{0,1\}^*$, as follows:

$$[\![h]\!]_{\mathscr{E}}(s) = s(h) \qquad [\![e[n_1 : n_2]]\!]_{\mathscr{E}}(s) = [\![e]\!]_{\mathscr{E}}(s)[n_1 : n_2]$$

$$[\![bv]\!]_{\mathscr{E}}(s) = bv \qquad [\![e_1 +\!+ e_2]\!]_{\mathscr{E}}(s) = [\![e_1]\!]_{\mathscr{E}}(s)[\![e_2]\!]_{\mathscr{E}}(s)$$

There is a straightforward typing judgement $\vdash_{\mathscr{E}}$, where $\vdash_{\mathscr{E}} e : n$ implies that $|[\![e]\!]_{\mathscr{E}}(s)| = n$. We elide this definition.

Next, we give a semantics to operations and transitions, which constitute the code that can appear inside states.

**Definition 4.3.2** (Operation Semantics)**.** When $s \in S$, $h \in H$ and $v \in \{0,1\}^{\mathsf{sz}(h)}$, we use $s[v/h]$ to denote the store where $s[v/h](h) = v$, and $s[v/h](h') = s(h')$ for all $h' \in H \setminus \{h\}$.

For operations *op*, define $\|op\| \in \mathbb{N}$ inductively, as follows:

$$\|h := e\| = 0 \qquad \|\texttt{extract}(h)\| = \mathsf{sz}(h)$$

$$\|op_1; op_2\| = \|op_1\| + \|op_2\|$$

Intuitively, $\|op\|$ is the exact number of bits necessary to execute all `extract` statements that appear in *op*.

For each block of operations *op*, we define a *partial* function $[\![op]\!]_{\mathscr{O}} : S \times \{0,1\}^* \rightharpoonup S \times \{0,1\}^*$, as follows:

$$[\![h := e]\!]_{\mathscr{O}}(s,w) = \langle s[v/h], w\rangle \qquad (\text{if } [\![e]\!]_{\mathscr{E}}(s) = v \text{ and } |v| = \mathsf{sz}(h))$$

$$[\![\texttt{extract}(h)]\!]_{\mathscr{O}}(s,xy) = \langle s[x/h], y\rangle \qquad (\text{if } |x| = \mathsf{sz}(h))$$

$$[\![op_1; op_2]\!]_{\mathscr{O}}(s,w) = [\![op_2]\!]_{\mathscr{O}}([\![op_1]\!]_{\mathscr{O}}(s,w))$$

There exists a type judgement $\vdash_{\mathscr{O}}$ such that if $\vdash_{\mathscr{O}} op$ then $[\![op]\!]_{\mathscr{O}}(s,w) \in S \times \{\epsilon\}$ for all $bv \in \{0,1\}^{\|op\|}$.

**Definition 4.3.3** (Pattern and Transition Semantics)**.** For a pattern *pat*, define $[\![pat]\!]_{\mathscr{P}} \subseteq \{0,1\}^*$ by case distinction:

$$[\![bv]\!]_{\mathscr{P}} = \{bv\} \qquad [\![\_]\!]_{\mathscr{P}} = \{0,1\}^*$$

Given a transition block *tz*, we define a partial function $[\![tz]\!]_{\mathscr{T}} : S \rightharpoonup Q \cup \{\mathsf{accept}, \mathsf{reject}\}$ inductively, as follows:

$$[\![\texttt{goto}(q)]\!]_{\mathscr{T}}(s) = q \qquad [\![\texttt{select}(\overline{e})\{\}]\!]_{\mathscr{T}}(s) = \mathsf{reject}$$

$$\frac{\forall i.\ [\![e_i]\!]_{\mathscr{E}}(s) = v_i \qquad q' = [\![\texttt{select}(\overline{e})\{\overline{c}\}]\!]_{\mathscr{T}}(s)}{[\![\texttt{select}(\overline{e})\{\overline{pat} \Rightarrow q;\ \overline{c}\}]\!]_{\mathscr{T}}(s) = \begin{cases} q & \forall i.\ v_i \in [\![pat_i]\!]_{\mathscr{P}} \\ q' & \text{otherwise} \end{cases}}$$

As before, a straightforward type judgement $\vdash_{\mathscr{T}}$ can be formulated such that $\vdash_{\mathscr{T}} tz$ implies that $[\![tz]\!]_{\mathscr{T}}(s)$ is defined.

We now have the ingredients necessary to define the dynamics of a P4A in terms of a deterministic finite automaton (DFA). To facilitate the comparison of P4As that consume packets in differently-sized chunks, this DFA buffers until it has read enough bits to execute the `extract` blocks associated with the current state. We first precisely define a configuration of a P4A, as follows.

**Definition 4.3.4** (Configurations)**.** A *configuration* is a triple

$$\langle q, s, w\rangle \in (Q \cup \{\mathsf{accept}, \mathsf{reject}\}) \times S \times \{0,1\}^*.$$

where $|w| < \|op(q)\|$ if $q \in Q$, and $w = \epsilon$ otherwise. We write $C$ for the (finite) set of configurations, and $F$ for the *accepting configurations*: $\{\langle \mathsf{accept}, s, \epsilon\rangle \in C : s \in S\}$.

We can define a bit-by-bit step function on configurations, which implements the idea of filling up the store before actuating the transition, outlined above.

**Definition 4.3.5** (Configuration Dynamics)**.** We define the step function $\delta : C \times \{0,1\} \rightharpoonup C$ as follows. Let $c = \langle q, s, w\rangle \in C$. If $q \in Q$, then we define $\delta(c, b)$ by setting

$$\delta(c,b) = \begin{cases} \langle q, s, wb\rangle & |wb| < \|op(q)\| \\ \langle [\![tz(q)]\!]_{\mathscr{T}}(s'), s', \epsilon\rangle & [\![op(q)]\!]_{\mathscr{O}}(s, wb) = \langle s', \epsilon\rangle \end{cases}$$

Otherwise, if $q \in \{\mathsf{accept}, \mathsf{reject}\}$, then $\delta(c, b) = \langle \mathsf{reject}, s, \epsilon\rangle$.

There exists a type judgement $\vdash_{\mathscr{A}}$ such that $\vdash_{\mathscr{A}} aut$ implies that $\delta$ is well-defined and total; again, we omit its definition.[4]

To match the behavior of P4 parsers, accepting states should not parse any further input. As a consequence a configuration of the form $\langle \mathsf{accept}, s, \epsilon\rangle$ steps unconditionally to $\langle \mathsf{reject}, s, \epsilon\rangle$.

Put together, $\langle C, \delta, F\rangle$ is a DFA. We can therefore define the language semantics of our parser *aut* as a function $[\![aut]\!]_{\mathscr{A}} : Q \times S \to 2^{\{0,1\}^*}$, where $2^X$ denotes the set of subsets of a set $X$. This semantics associates with each initial state and store the set of bit-strings that lead to an accepting configuration.

[4] Our requirement that each state extracts some bits is part of this typing judgement, and in fact necessary in order for the definition of $\delta$ to be useful. Because a transition is triggered by the final bit, if $\|op(q)\| = 0$ for some state, then there would be no way to actuate this transition.

**Definition 4.3.6** (Multi-Step Configuration Dynamics)**.** We can lift $\delta$ to $\delta^* : C \times \{0,1\}^* \rightharpoonup C$ as follows:

$$\delta^*(c, \epsilon) = c \qquad\qquad \delta^*(c, bw) = \delta^*(\delta(c, b), w)$$

Given $c \in C$, we define its *language* $L(c) \subseteq \{0,1\}^*$ as follows:

$$L(c) = \{w \in \{0,1\}^* : \delta^*(c, w) \in F\}$$

Given $q \in Q$ and $s \in S$, we define $[\![aut]\!]_{\mathscr{A}}(q, s) = L(q, s, \epsilon)$.

Our semantics embeds the initial store in the start state. Our equivalence checking procedure can help verify that packet acceptance does not depend on the initial store value.

## 4.4 Symbolic Equivalence Checking

Many verification questions about P4As can be phrased as questions about the underlying DFAs. For instance, let $aut_1$ and $aut_2$ be the P4As from Figure 4.1, suppose we want to verify that they accept the same packets when started from certain initial states `q1` and `q3`, regardless of their initial store. To do this, we could check whether $L(\texttt{q1}, s_1, \epsilon) = L(\texttt{q3}, s_2, \epsilon)$ for all $s_1, s_2 \in S$. This problem is decidable, because $S$ is finite and language equivalence of DFAs is decidable [79].

Unfortunately, the DFA arising from a P4A may be extremely large: every $q \in Q$ contributes $|S| \times 2^{\|op(q)\|-1}$ configurations. Even for simple parsers, this leads to an intractably large configuration space. For instance, for the reference MPLS parser on the left in Figure 4.1, $|S| = 2^{96}$; a back-of-the-envelope calculation then tells us that $|C| \geq 10^{38}$.

Moreover, we anticipate that a large portion of the configuration space is reachable, and should therefore be taken into consideration. This is because parsers tend to propagate every bit of the packet into the store in order to facilitate packet reconstruction

for forwarding. Off-the-shelf algorithms for DFAs are therefore unlikely to scale to this setting.

In this section, we develop an algorithm that can answer several questions about P4As. This algorithm mitigates state space explosion by representing configurations symbolically. Our presentation focuses on deciding language equivalence of configurations. As a consequence, the procedure can be thought of as a variation on Moore's algorithm [79]. We discuss more general applications in Section 4.7.

For the sake of simplicity, we fix a P4A *aut* with underlying DFA $\langle C, \delta, F \rangle$ for the remainder of this section. One can compare configurations in two different P4As by taking their disjoint sum, renaming states and headers as necessary.

### 4.4.1 A Symbolic Approach

A sound and complete method to show that two configurations of our DFA $\langle C, \delta, F \rangle$ accept the same language is to demonstrate that they are related by a *bisimulation* [56], i.e., a relation $R \subseteq C \times C$ such that when $c_1 \; R \; c_2$, (1) $c_1 \in F$ if and only if $c_2 \in F$; and (2) $\delta(c_1, b) \; R \; \delta(c_2, b)$ for all $b \in \{0,1\}$.

A language equivalence checking algorithm for DFAs typically tries to build some form of bisimulation. Because $C$ may be very large, representing a bisimulation by listing its constituent pairs becomes intractable quickly. Luckily, we can write down bisimulations symbolically.

**Example 4.4.1.** Suppose *aut* is the disjoint sum of the MPLS parsers displayed in Figure 4.1. Let $R$ be the smallest relation on $C$ satisfying the following rules for all $s_1, s_2 \in S$:

$$\frac{w \in \{0,1\}^{32} \quad x \in \{0,1\}^* \quad |x| < 32}{\langle \mathsf{q2}, s_1, wx \rangle \; R \; \langle \mathsf{q5}, s_2, x \rangle} \qquad \frac{q \in \{\mathsf{accept}, \mathsf{reject}\}}{\langle q, s_1, \epsilon \rangle \; R \; \langle q, s_2, \epsilon \rangle}$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $x$ | $\in$ | $\mathsf{Var}$ | variables | $p$ | $::=$ | $be_1 = be_2$ | bitvector equality |
| $be$ | $::=$ | $bv$ | literal | | $\mid$ | $q^<, q^>$ | left and right state assertion |
| | $\mid$ | $\mathrm{buf}^<, \mathrm{buf}^>$ | left and right buffer | | $\mid$ | $n^<, n^>$ | left and right buffer length |
| | $\mid$ | $h^<, h^>$ | left and right header | $\phi$ | $::=$ | $\bot$ | bottom |
| | $\mid$ | $x$ | variable | | $\mid$ | $p$ | atomic predicate |
| | $\mid$ | $be[n_1 : n_2]$ | slice | | $\mid$ | $\phi_1 \implies \phi_2$ | implication |
| | $\mid$ | $be_1 +\!\!+ be_2$ | concat | | | | |

Figure 4.3: Syntax for relations on configurations.

$R$ is a bisimulation, and thus all configurations related by $R$ have the same language. Clearly, this representation is much more concise than listing the contents of $R$ explicitly.

To systematically represent and manipulate symbolic relations on configurations, we propose the syntax in Figure 4.3. Its formulas are generated by equality assertions between expressions built over the buffers and stores of both configurations, as well as predicates about states and buffer lengths. We also include variables $x \in \mathsf{Var}$ for later use. We omit conjunction ($\vee$) and disjunction ($\wedge$) from the syntax to keep our definitions brief. They are derivable from $\implies$ and $\bot$, so we will still use them in the sequel as abbreviations.

**Example 4.4.2.** We can describe the pairs matching the second rule from Example 4.4.1 using the following formula

$$\phi = (\mathsf{accept}^< \wedge \mathsf{accept}^>) \vee (\mathsf{reject}^< \wedge \mathsf{reject}^>)$$

Given $n < 32$, we can choose the formula $\phi_n$ to symbolize the first rule from Example 4.4.1 where $|x| = n$:

$$\phi_n = (n+32)^< \wedge q2^< \wedge n^> \wedge q5^> \wedge \mathrm{buf}^<[0{:}31] = \mathrm{buf}^>$$

In total, $R$ is represented by the formula $\phi \vee \phi_0 \vee \cdots \vee \phi_{31}$.

**Definition 4.4.3.** A *valuation* is a function $\sigma : \mathsf{Var} \to \{0,1\}$. For every bitvector expression $be$ and valuation $\sigma$, we define $[\![be]\!]^{\sigma}_{\mathscr{B}} : C \times C \to \{0,1\}^*$ inductively as follows, where $c^<, c^> \in C$ are such that $c^{\lessgtr} = \langle q^{\lessgtr}, s^{\lessgtr}, w^{\lessgtr} \rangle$ for $\lessgtr \in \{<, >\}$:

$$[\![bv]\!]^{\sigma}_{\mathscr{B}}(c^<, c^>) = bv \qquad [\![\mathrm{buf}^{\lessgtr}]\!]^{\sigma}_{\mathscr{B}}(c^<, c^>) = w^{\lessgtr} \qquad [\![x]\!]^{\sigma}_{\mathscr{B}}(c^<, c^>) = \sigma(x)$$

$$[\![h^{\lessgtr}]\!]^{\sigma}_{\mathscr{B}}(c^<, c^>) = s^{\lessgtr}(h)$$

The cases for slices and concatenation are as in Definition 4.3.1.

For a formula $\phi$ and valuation $\sigma$, define $[\![\phi]\!]^{\sigma}_{\mathscr{L}}$ as the least relation on $C$ satisfying the following rules for all $c^<, c^> \in C$, where $c^{\lessgtr} = \langle q^{\lessgtr}, s^{\lessgtr}, w^{\lessgtr} \rangle$, and $n^{\lessgtr} = |w^{\lessgtr}|$ for $\lessgtr \in \{<, >\}$:

$$c^< \; [\![q^<]\!]^{\sigma}_{\mathscr{L}} \; c^> \qquad \frac{[\![be_2]\!]_{\mathscr{B}}(c_1, c_2, \sigma) = [\![be_2]\!]_{\mathscr{B}}(c_1, c_2, \sigma)}{c_1 \; [\![be_1 = be_2]\!]^{\sigma}_{\mathscr{L}} \; c_2}$$

$$c^< \; [\![q^>]\!]^{\sigma}_{\mathscr{L}} \; c^>$$

$$c^< \; [\![n^<]\!]^{\sigma}_{\mathscr{L}} \; c^> \qquad \frac{c_1 \; [\![\phi_1]\!]^{\sigma}_{\mathscr{L}} \; c_2 \implies c_1 \; [\![\phi_2]\!]^{\sigma}_{\mathscr{L}} \; c_2}{c_1 \; [\![\phi_1 \implies \phi_2]\!]^{\sigma}_{\mathscr{L}} \; c_2}$$

$$c^< \; [\![n^>]\!]^{\sigma}_{\mathscr{L}} \; c^>$$

Let $\phi$ and $\psi$ be formulas. We write $[\![\phi]\!]_{\mathscr{L}}$ for the relation on $C$ where $c_1 \; [\![\phi]\!]_{\mathscr{L}} \; c_2$ if and only if $c_1 \; [\![\phi]\!]^{\sigma}_{\mathscr{L}} \; c_2$ for all valuations $\sigma$. Finally, we write $\phi \vDash \psi$ when $[\![\phi]\!]_{\mathscr{L}} \subseteq [\![\psi]\!]_{\mathscr{L}}$.

Note that because there are finitely many configurations and valuations, entailments are decidable. We will revisit this particular decision problem in Sections 4.5 and 4.6.

We can now define symbolic bisimulations, as follows.

**Definition 4.4.4** (Symbolic Bisimulation)**.** A *symbolic bisimulation* is a formula $\phi$ such that $[\![\phi]\!]_{\mathscr{L}}$ is a bisimulation.

Finding a (symbolic) bisimulation is a sound and complete method to establish language equivalence of states.

**Algorithm 1:** Symbolic equivalence checking.

**Input:** A formula $\phi$ representing initial states.
**Input:** A set of formulas $I$ s.t. for all $c_1, c_2 \in C$,

$$\left[\forall \psi \in I.\ c_1 \ [\![\psi]\!]\ c_2\right] \Leftrightarrow [c_1 \in F \Leftrightarrow c_2 \in F]$$

**Input:** A function WP s.t. for all $\psi$, and $c_1, c_2 \in C$,

$$\left[\forall b \in \{0,1\}.\ \delta(c_1, b)\ [\![\psi]\!]\ \delta(c_2, b)\right] \Leftrightarrow c_1\ [\![\textstyle\bigwedge \mathsf{WP}(\psi)]\!]\ c_2$$

**Output: true** if and only if for all $c_1, c_2 \in C$ with $c_1\ [\![\phi]\!]_{\mathscr{L}}\ c_2$, it holds that $L(c_1) = L(c_2)$

```
1 R ← ∅; T ← I
2 while T ≠ ∅ do
3 |   pop ψ from T
4 |   if not ⋀R ⊨ ψ then
5 |   |   R ← R ∪ {ψ}
6 |   |   T ← T ∪ WP(ψ)
7 return true if φ ⊨ ⋀R, otherwise false
```

**Lemma 4.4.1.** *For formulas $\phi$, the following are equivalent:*

1. *There exists a symbolic bisimulation $\psi$ such that $\phi \models \psi$.*
2. *There exists a bisimulation $R$ such that $[\![\phi]\!]_{\mathscr{L}} \subseteq R$.*
3. *If $c_1\ [\![\phi]\!]_{\mathscr{L}}\ c_2$, then $L(c_1) = L(c_2)$.*

### 4.4.2 The Weakest Symbolic Bisimulation

To search for a symbolic bisimulation, we turn to Moore's algorithm [79]. In its concrete formulation, this algorithm approximates the largest (coarsest) bisimulation from above, by iteratively removing non-bisimilar pairs. Eventually, the process stops, at which point the remaining pairs must be bisimilar; hence, the computed relation is the largest bisimulation. Two configurations are related by some bisimulation if and only if they are related by the largest bisimulation, so the algorithm concludes with a simple containment check.

Moore's algorithm can be made symbolic, by representing the current overapproximation as a formula and successively strengthening it, thus converging to the weakest symbolic bisimulation. We present an abstract formulation of this process in Algorithm 1. The algorithm has two parameters.

- The formulas in $I$ constitute the initial overapproximation of the weakest symbolic bisimulation. In Section 4.7, we consider instantiations of $I$ that can be used to verify different but related properties.
- The function $\mathsf{WP}$ takes a formula $\phi$, and outputs a set of formulas whose conjunction represents a *weakest precondition* of $\phi$, in the sense that two configurations are related by *all* formulas in $\mathsf{WP}(\phi)$ if and only if they step into configurations related by $\phi$.

Algorithm 1 builds the weakest symbolic bisimulation as a set of conjuncts $R$, maintaining a frontier $T$ of formulas to be considered. The frontier is initially $I$. In each iteration, we pop a conjunct $\psi$ from $T$ and check if it is entailed by $\bigwedge R$. If $\bigwedge R \not\models \phi$, then $\psi$ constitutes a novel restriction, and we add it to $R$. Because bisimulations are closed under steps, we add the weakest preconditions of $\psi$ to $T$, to be checked later. If $\bigwedge R \models \psi$, including $\psi$ in $R$ would not change $\bigwedge R$, so we move on. The loop terminates when $T$ is empty; at this point, $\bigwedge R$ will be the weakest symbolic bisimulation, and the algorithm checks $\phi \models \bigwedge R$. We instantiate the parameters momentarily; first, we show that the algorithm is correct.

**Theorem 4.4.5.** *Algorithm 1 is correct.*

*Proof Sketch.* For termination, note that in each iteration either $[\![\bigwedge R]\!]$ or $T$ shrinks; hence, the algorithm must terminate.

For partial correctness, one can show the following invariants: (1) if $\phi$ is a symbolic bisimulation, then $\phi \models \bigwedge R \wedge \bigwedge T$; and (2) configurations related by $\bigwedge R \wedge \bigwedge T$ are

equally accepting, and (3) configurations related by $\bigwedge R \wedge \bigwedge T$ step into configurations related by $\bigwedge R$. Thus, when Algorithm 1 terminates, $\bigwedge R$ must be the largest symbolic bisimulation, and we can conclude by applying Lemma 4.4.1. □

### 4.4.3 Instantiating the Parameters

We now sketch how we instantiate the parameters of Algorithm 1; the details are worked out in our Coq development.

For $\mathsf{WP}$, the main idea is to focus on a particular subclass of formulas. First, we isolate assertions about the current state; this lets us calculate weakest preconditions on a state-by-state basis, by means of a traditional substitution-based procedure on the formula using the associated program text. Second, we isolate statements about buffer lengths; this means that when a formula in our algorithm makes a claim about the buffer contents, it does so in a context where the buffer length is known. This simplifies the analysis and generation of formulas, because we do not have to cover cases where slices go beyond the end of a bitvector.

Concretely, this format takes the following form.

**Definition 4.4.6** (Templates)**.** A *template* is a pair $\langle q, n\rangle \in (Q \cup \{\mathsf{accept}, \mathsf{reject}\}) \times \mathbb{N}$ where $n < \|op(q)\|$ if $q \in Q$, and $n = 0$ otherwise. The set of all templates is $T$. When $t = \langle q, n\rangle$ and $\lessgtr \in \{<, >\}$, we write $t^{\lessgtr}$ as shorthand for $q^{\lessgtr} \wedge n^{\lessgtr}$.

A formula $\phi$ is *pure* when it does not contain state or buffer length assertions; $\phi$ is *template-guarded* if it is of the form $t_1^{<} \wedge t_2^{>} \implies \psi$ where $t_1, t_2 \in T$ and $\psi$ is pure.

Let $\phi$ be template-guarded. We compute $\mathsf{WP}(\phi)$ by operating on the left- and right hand side, giving rise to functions $\mathsf{WP}^{<}$ and $\mathsf{WP}^{>}$, whose definitions we omit. Each takes a formula and a pair of state templates, as well as a fresh variable $x \in \mathsf{Var}$ to represent the bit to be read, and returns a formula. The relevant correctness statement

is as follows.

**Lemma 4.4.2.** *Let $\phi$ be pure, let $x \in \mathsf{Var}$ be fresh for $\phi$, and let $c^<, c^> \in C$ as well as $t \in T$. The following are equivalent:*

1. *For all $b \in \{0,1\}$, we have $\delta(c^<, b)\ [\![ t^< \implies \phi ]\!]_{\mathscr{L}}\ c^>$.*
2. *For all $t' \in T$, $c^<\ [\![ t'^< \implies \mathsf{WP}^<(\phi, t', t, x) ]\!]_{\mathscr{L}}\ c^>$.*

*A similar equivalence holds for $\mathsf{WP}^>$. Furthermore, if $\phi$ is pure, then so are $\mathsf{WP}^<(\phi, x, t', t)$ and $\mathsf{WP}^>(\phi, x, t', t)$.*

Using $\mathsf{WP}^<$ and $\mathsf{WP}^>$, we can then provide a version of $\mathsf{WP}$ that acts on and returns template-guarded formulas. Its definition and correctness statement is as follows.

**Lemma 4.4.3.** *Let $t_1^< \wedge t_2^> \implies \phi$ be template-guarded, and let $x$ be fresh in $\phi$. Define $\mathsf{WP}(t_1^< \wedge t_2^> \implies \phi)$ as the smallest set satisfying the following rule, for all $t_1', t_2' \in T$:*

$$\frac{\phi' = \mathsf{WP}^<(\mathsf{WP}^>(\phi', x, t_2', t_2), x, t_1', t_1)}{[t_1'^< \wedge t_2'^> \implies \phi'] \in \mathsf{WP}(t_1^< \wedge t_2^> \implies \phi)}$$

*Now $\mathsf{WP}$ fits the requirement from Algorithm 1, when restricted to template-guarded formulas. Moreover, each of the formulas in $\mathsf{WP}(t_1^< \wedge t_2^> \implies \phi)$ is template-guarded.*

By the latter property, if all formulas in $I$ are template-guarded, then the formulas in $R$ and $T$ remain template-guarded. We thus instantiate $I$ as a set of template-guarded formulas that rule out pairs containing both accepting and non-accepting configurations, as follows.

**Lemma 4.4.4.** *Let $t_{\mathsf{accept}} = \langle \mathsf{accept}, 0 \rangle$. Define $I$ as the smallest set of formulas satisfying the following rule:*

$$\frac{t_1, t_2 \in T \qquad t_1 = t_{\mathsf{accept}} \iff t_2 \neq t_{\mathsf{accept}}}{[t_1^< \wedge t_2^> \implies \bot] \in I}$$

*Now $I$ fits the requirement from Algorithm 1.*

## 4.5 Optimizing the Algorithm

We now discuss two optimizations of Algorithm 1. The first optimization refines WP and $I$ such that fewer entailments between formulas need to be checked (line 4). The second optimization generalizes WP to compute multi-step weakest preconditions, thereby strengthening the approximation of the weakest symbolic bisimulation more quickly.

### 4.5.1 Abstract Interpretation

Algorithm 1 computes the weakest symbolic bisimulation, which relates *all* language equivalent configurations, but it cares only about the configurations related by $\phi$. We can compute a symbolic bisimulation more loosely, by disregarding unreachable (and hence, irrelevant) configuration pairs.

**Example 4.5.1.** Recall the symbolic bisimulation in Example 4.4.2, which was sufficient to conclude language equivalence of related configurations. There was no need to compute the largest symbolic bisimulation, which involves many configuration pairs unreachable from the pairs of interest.

Of course, computing the set of reachable pairs—even symbolically—is tantamount to checking equivalence. Instead, we approximate it by analyzing the P4A to capture the pairs of reachable configurations based on their templates.

To this end, let $\rho(tz)$ denote the set of states appearing in a transaction block $tz$. We define $\sigma : T \to 2^T$ as follows:

$$\sigma(q,n) = \begin{cases} \{\langle q, n+1\rangle\} & q \in Q \wedge n+1 < \mathsf{sz}(q) \\ \rho(tz(q)) \times \{0\} & q \in Q \wedge n+1 = \mathsf{sz}(q) \\ \{\langle \mathsf{reject}, 0\rangle\} & q \in \{\mathsf{accept}, \mathsf{reject}\} \end{cases}$$

When $c = \langle q, s, w\rangle \in C$, write $\lfloor c \rfloor$ for $\langle q, |w|\rangle \in T$, i.e., the unique template describing $c$. One can show that for all $c \in C$ and $b \in \{0,1\}$, we have $\lfloor \delta(c,b) \rfloor \in \sigma(\lfloor c \rfloor)$. In a sense, this makes $\sigma$ an abstract interpretation of $\delta$.

Given a formula $\phi$, we define $\mathsf{reach}_\phi$ as the smallest relation on $T$ satisfying the following rules:

$$\frac{c_1 \; [\![\phi]\!]_{\mathscr{L}} \; c_2}{\lfloor c_1 \rfloor \; \mathsf{reach}_\phi \; \lfloor c_2 \rfloor} \qquad \frac{t_1 \; \mathsf{reach}_\phi \; t_2}{\sigma(t_1) \times \sigma(t_2) \subseteq \mathsf{reach}_\phi}$$

Usually, the pairs generated by the first rule can be inferred from $\phi$. For instance, if we want to compare the languages of two initial states $q_1$ and $q_2$, then $\phi = q_1^< \wedge 0^< \wedge q_2^> \wedge 0^>$, and so the sole instantiation of the first rule yields $\langle q_1, 0\rangle \; \mathsf{reach}_\phi \; \langle q_2, 0\rangle$. Computing the full contents of $\mathsf{reach}_\phi$ is then a matter of applying the second rule until a fixpoint is reached.

**Theorem 4.5.2.** *Let $\phi$ be a formula. Algorithm 1 remains correct for this $\phi$ if we set $I$ to the smallest set satisfying the rule*

$$\frac{t_1 \; \mathsf{reach}_\phi \; t_2 \qquad t_1 = t_{\mathsf{accept}} \iff t_2 \neq t_{\mathsf{accept}}}{[t_1^< \wedge t_2^> \implies \bot] \in I}$$

*and for each template-guarded formula $t_1^< \wedge t_2^> \implies \psi$ we set $\mathsf{WP}(t_1^< \wedge t_2^> \implies \psi)$ to the smallest set satisfying the rule*

$$\frac{t_1' \; \mathsf{reach}_\phi \; t_2' \qquad \psi' = \mathsf{WP}^<(\mathsf{WP}^>(\psi', x, t_2', t_2), x, t_1', t_1)}{[t_1'^< \wedge t_2'^> \implies \phi'] \in \mathsf{WP}(t_1^< \wedge t_2^> \implies \phi)}$$

*where $x \in \mathsf{Var}$ is some variable that is fresh for $\psi$.*

### 4.5.2 Leaps and Bounds

Algorithm 1 operates on a bit-by-bit basis. However, most steps just fill up the buffer, and do not affect the state or store. We exploit this to compute a different form of

weakest precondition, which takes as many steps as necessary to execute a "real" state-to-state transition in the P4A.

The following auxiliary notion allows us to compute the number of steps until the next transition.

**Definition 4.5.3** (Leap Size)**.** Let $c_1, c_2 \in C$ and $c_i = \langle q_i, s_i, w_i \rangle$; we define the *leap size* $\sharp(c_1, c_2) \in \mathbb{N}$ as follows:

$$\sharp(c_1, c_2) = \begin{cases} 1 & q_1, q_2 \notin Q \\ \|tz(q_1)\| - |w_1| & q_1 \in Q, q_2 \notin Q \\ \|tz(q_2)\| - |w_2| & q_1 \notin Q, q_2 \in Q \\ \min(\|tz(q_1)\| - |w_1|, \\ \qquad \|tz(q_2)\| - |w_2|) & q_1, q_2 \in Q \end{cases}$$

We can use leap size to define a notion of (symbolic) bisimilarity that can take larger steps; this will help us to formally justify the soundness of multi-step weakest preconditions.

**Definition 4.5.4** (Bisimulation with Leaps)**.** A *bisimulation with leaps* is a relation $R \subseteq C \times C$, such that for all $c_1\ R\ c_2$, (1) $c_1 \in F$ if and only if $c_2 \in F$, and (2) $\delta^*(c_1, w)\ R\ \delta^*(c_2, w)$ for all $w \in \{0,1\}^{\sharp(c_1,c_2)}$. A *symbolic bisimulation with leaps* is a formula $\phi$ such that $[\![\phi]\!]_{\mathscr{L}}$ is a bisimulation with leaps.

Bisimulations with leaps can be more concise because they do not need to constrain configurations where both P4A are just buffering input, waiting for the next transition.

**Example 4.5.5.** Recall the bisimulation from Example 4.4.1. This relation contains the

bisimulation with leaps $R'$, which is the smallest relation satisfying the rules

$$\frac{w \in \{0,1\}^{32}}{\langle \mathsf{q2}, s_1, w\rangle \; R' \; \langle \mathsf{q5}, s_2, \epsilon\rangle} \qquad \frac{q \in \{\mathsf{accept}, \mathsf{reject}\}}{\langle q, s_1, \epsilon\rangle \; R' \; \langle q, s_2, \epsilon\rangle}$$

Bisimilarity with leaps is a sound and complete proof principle for language equivalence, which we record as follows.

**Lemma 4.5.1.** *Let $\phi$ be a formula. The following are equivalent:*

1. *There exists a symbolic bisim. with leaps $\psi$ s.t. $\phi \vDash \psi$.*
2. *If $c_1 \; [\![\phi]\!]_{\mathscr{L}} \; c_2$, then $L(c_1) = L(c_2)$.*

We can adapt Algorithm 1 to calculate the weakest symbolic bisimulation with leaps instead, if we adapt the axiomatization of the weakest precondition operator, as follows.

**Theorem 4.5.6.** *Algorithm 1 remains correct if we change the condition on* WP *to require that, for all formulas $\phi$ and all $c_1, c_2 \in C$, the following equivalence holds:*

$$\begin{aligned} &\forall w \in \{0,1\}^{\sharp(c_1,c_2)}.\; \delta^*(c_1, w) \; [\![\phi]\!]_{\mathscr{L}} \; \delta^*(c_2, w) \\ &\iff \forall \psi \in \mathsf{WP}(\phi).\; c_1 \; [\![\psi]\!]_{\mathscr{L}} \; c_2 \end{aligned}$$

We can adapt the existing definition of WP to conform to this specification: simply repeat $\mathsf{WP}^<$ and $\mathsf{WP}^>$ as many times as is indicated by the source templates $t_1'$ and $t_2'$.

### 4.5.3 Combining Optimizations

The optimizations discussed are largely orthogonal. However, their combination naturally gives rise to a third optimization, where $\mathsf{reach}_\phi$ is computed using leaps as well. This results in an algorithm that computes a symbolic bisimulation with leaps that does not constrain intermediate (buffering) configurations. We refer to the Coq development for full details.

Table 4.1: Concepts from earlier in this chapter and their realizations in the implementation.

| Paper name | Coq name | Implemented as |
|---|---|---|
| *aut* (Figure 4.2) | `Syntax.t` | Dependent record |
| $e$ (Figure 4.2), $\vdash_{\mathscr{E}}$ | `expr` | Type-indexed ind. |
| WP | `wp` | Gallina function |
| $\bigwedge R \vDash \psi$ | `interp_entailment` | Gallina function |
| $\phi \vDash \bigwedge R$ | `interp_entailment'` | Gallina function |
| Bisimilarity | `bisimilar` | Coinductive relation |
| Algorithm 1 | `pre_bisimulation` | Inductive relation |
| **if** $\bigwedge R \vDash \psi \ldots$ | `decide_entailment` | $\mathrm{L}_{\mathrm{tac}}$ |

## 4.6 Implementation

We implement Leapfrog in Coq [12, 25] using the Equations plugin [99, 100]. See Table 4.1 for a summary of how concepts from the formal development in Sections 4.3 to 4.5 map to Coq notions. Although an implementation in a different language might be more efficient, our use of Coq produces rich semantic automata definitions and reusable proofs of equivalence. These artifacts are defined in Coq's expressive higher-order logic, so they can be reused and composed with other mechanized logics hosted in Coq like the Mathematical Components library [74] or verification tools like the Verified Software Toolchain [4].

### 4.6.1 Automated Proof Search

The most direct way to implement algorithms in Coq is by writing them as functions in Gallina, Coq's functional programming language, but unfortunately Gallina does not have I/O. As a consequence a Gallina implementation of Algorithm 1 would have to include a hand-written decision procedure for entailments $\bigwedge R \vDash \psi$. We instead realize Algorithm 1 in Coq as an inductive relation (Figure 4.4), so we can rely on external SMT solvers to handle entailments. This has the added benefit of sidestepping Coq's

```
Inductive pre_bisimulation
  : conf_rel → list conf_rel → list conf_rel → Prop :=
| Skip: forall phi R psi T,
    pre_bisimulation phi R T →
    interp_entailment R psi →
    pre_bisimulation phi R (psi :: T)
| Extend: forall phi R psi T,
    pre_bisimulation (psi :: R) (T ++ wp psi) →
      interp_entailment R psi →
    pre_bisimulation R (psi :: T)
| Done: forall phi R,
    interp_entailment' phi R →
    pre_bisimulation phi R [].
```

Figure 4.4: Algorithm 1 as an inductive relation in Coq.

termination checker.[5] The algorithm is run by performing proof search within the inductive relation, and each step of the search proceeds by checking an entailment in the high-level automata logic. While the logic of entailments is close to SMT's theory of bitvectors, it also has richer terms that need to be desugared (for example a finite-map encoding of the program store, constraints on the input packet length, constraints on the automata states, etc.).

### 4.6.2 Reduction to SMT

To reach a low-level logic amenable to off-the-shelf solvers, we simplify formulas before checking them, through a chain of verified simplifications and translations (Figure 4.6).

This compilation turns formulas from the high-level logic ConfRel into low-level first-order formulas over bitvectors, FOL(BV). In order, the implementation performs (1) algebraic simplifications, (2) template filtering, (3) FOL compilation, and (4) store elimination. We now elaborate on each step.

First, we use smart constructors to apply local algebraic simplifications. Each

[5] In particular, our pen-and-paper termination proof of Algorithm 1 does not directly translate to Coq's guarded primitive recursion [45].

```
Lemma small_filter_equiv:
  lang_equiv_state
    (P4A.interp IncrementalBits.aut)
    (P4A.interp BigBits.aut)
    IncrementalBits.Start
    BigBits.Parse.
Proof.
  solve_lang_equiv_state_axiom
    IncrementalBits.state_eqdec
    BigBits.state_eqdec
    false.
Time Qed.
```

Figure 4.5: A Coq proof that the states `Start` and `Parse` of two automata named `IncrementalBits` and `BigBits` accept the same language (`lang_equiv_state`). The tactic `solve_lang_equiv_state_axiom` takes decision procedures for equality on the state sets of each automaton and a flag controlling a tactic optimization for large problems (here `false`).

application of the weakest precondition operator increases the size of a formula, so these simplifications help prevent the formulas from growing too quickly.

Second, we perform template filtering to discard unused premises from entailments. Entailments have the form

$$\bigwedge \phi_i \implies \psi_i \vDash \phi \implies \psi,$$

where $\phi$ and all $\phi_i$ are templates. We discard any conjunct with $\phi_i \neq \phi$ and emit a simplified entailment $\phi \vDash \bigwedge \psi_i \implies \psi$. This puts our goal in the logic ConfRelSimp.

Third, we embed ConfRelSimp into the more general FOL(Conf) syntax, removing references to states. This fragment is the first-order theory of bitvectors and finite maps.

Finally, the store elimination pass fits formulas into the theory of bitvectors FOL(BV), by turning finite maps into first-order variables. This is necessary because some SMT solvers we targeted do not support the theory of finite maps.

### 4.6.3 Querying Solvers

The final FOL(BV) formula is serialized to SMT-LIB by a custom Coq plugin and passed to an off-the-shelf SMT solver that can be selected using a custom vernacular command. Currently, we support Z3 [30], CVC4 [9], and Boolector [83].

Before implementing our own plugin, we tried existing SMT integrations for Coq, including CoqHammer [27] and SMTCoq [7]. Neither solved our problem: CoqHammer scaled poorly due to its flexible SMT encoding and proof search procedure, while SMTCoq performed better but lacked support for quantifiers. Note however that, in contrast with our plugin, both of these tools perform *proof reconstruction* to produce a Coq proof term from solver output. Consequently, our proof search must *trust* the output of the SMT solver and our plugin. A straightforward technique for doing this is to directly `admit` the low-level goals once the plugin has given the thumbs up. This is rather error-prone because it means `admit` is used within automation, and moreover, it forces the final proof to be `Admitted` by the Coq kernel. An alternative is to use a pair of *axioms* for positive and negative validity of formulas in the low-level logic and use the output of the SMT solver to conditionally apply the axioms. While this approach is less performant because the Coq kernel checks the resulting term, it allows for closed proof terms and avoids accidental misuse of `admit` in automation.

### 4.6.4 Soundness and Trusted Computing Base

The most important metatheoretic goal is to ensure that our algorithm produces a certificate of equivalence only when the input parsers are indeed equivalent. Towards this goal, our certificate-producing equivalence checker has a compact TCB, with soundness relying on the Coq definitions of automata and automata equivalence, the correctness of the SMT solver, the faithfulness of the pretty-printing plugin, and the

soundness of the Coq typechecker extended with Streicher's axiom K [102]. The SMT solver and plugin (and the corresponding use of `admit`/axioms) are used only in the proof search algorithm and could be removed from the TCB by implementing proof reconstruction.

Our Coq development proves the soundness theorems stated in the paper, but omits completeness and termination arguments. Our proof search is really a semi-decision procedure: either the tactic finds a proof and produces a Coq proof term, or it does not find a proof and no certificate is produced. Trustworthy certificates, our main metatheoretic goal, only require a mechanization of soundness. In fact, the only termination or completeness bug we encountered arose from incorrectly interpreting failed SMT queries as UNSAT, which was a bug in the plugin and not in the algorithm itself.

## 4.7 Evaluation

We evaluate Leapfrog through case studies (listed in Table 4.2) along two dimensions: (1) the *utility* of bisimulations for solving problems of interest in networking (and, by extension, the *expressiveness* of Leapfrog), and (2) the *applicability* of Leapfrog to real-world parsers (and, by extension, its *scalability* to non-trivial inputs).

### 4.7.1 Utility

To evaluate whether equivalence checks are useful in the networking domain, we identified *six* distinct verification tasks, and showed how they can be solved with Leapfrog.

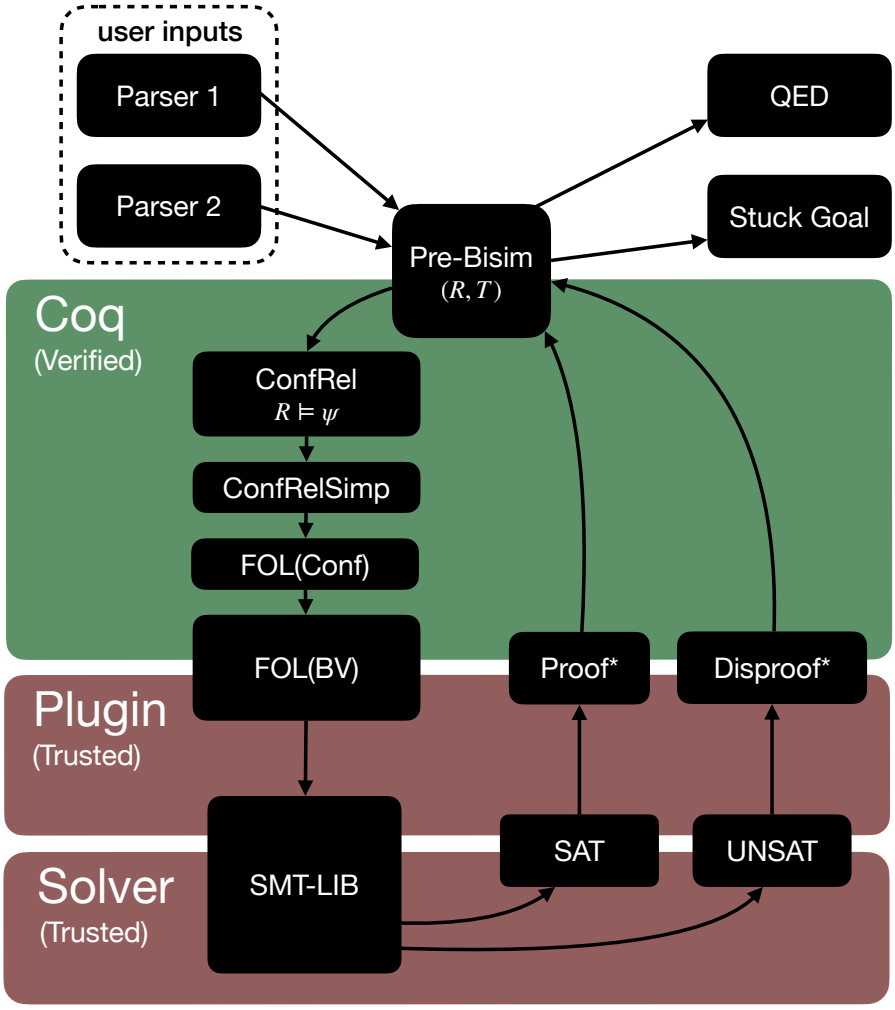



Figure 4.6: The Leapfrog implementation architecture. In each iteration, a ConfRel formula is checked by reduction to SMT via a chain of intermediate logics (at left). A Coq plugin pretty-prints FOL(BV) syntax to SMT-LIB syntax and invokes the SMT solver. The asterisk (*) on Proof and Disproof (at right) indicates that the plugin does not produce proofs. When the procedure halts, either the Coq goal is provable (QED), or the goal is stuck and no certificate is produced.

**State Rearrangement.** Because parser states translate to hardware resources, it is common for compilers to merge and split parser states, to optimize the write and branch behavior for the particular hardware. We implemented a reference parser for a stylized IP and UDP/TCP protocol in which the prefix is 64 bits of IP and the suffix is either 32 bits of UDP or 64 bits of TCP (Figure 4.7). Note that the TCP and UDP headers share a common prefix of 32 bits. We then implemented an optimized parser that extracts the IP and common prefix, and then branches to determine how to parse the

Table 4.2: Parsers in our evaluation: **States** gives the total number of states in both parsers, **B**ranched gives the number of bits in automata transition select statements, **T**otal gives the number of bits across all variables, and **Runtime** and **Mem**ory give the aggregate runtime and maximum resident size. An optimal verification algorithm would need to represent $2^{\mathbf{B}}$ states, while an explicit state space would contain $2^{\mathbf{T}}$ states. An asterisk (*) on the memory use indicates an out-of-memory exception.

| | **Name** | **States** | **B** (bits) | **T** (bits) | **Runtime** (min) | **Mem** (GB) |
|---|---|---|---|---|---|---|
| **Utility** | State Rearrangement | 5 | 8 | 136 | 0.12 | 0.66 |
| | Variable-length parsing | 30 | 64 | 632 | 953.42 | 405.64 |
| | Header initialization | 10 | 10 | 320 | 15.95 | 13.71 |
| | Speculative loop | 5 | 2 | 160 | 4.12 | 3.16 |
| | Relational verification | 6 | 64 | 1056 | 1.68 | 2.07 |
| | External filtering | 6 | 64 | 1056 | 1.18 | 1.71 |
| **Applicability** | Edge | 28 | 52 | 3184 | 528.38 | 251.26 |
| | Service Provider | 22 | 50 | 2536 | 1244.5 | 499.80* |
| | Datacenter | 30 | 242 | 2944 | 1387.95 | 404.50 |
| | Enterprise | 22 | 176 | 2144 | 217.93 | 66.13 |
| | Translation Validation | 30 | 56 | 3148 | 746.2 | 350.48 |

remaining bits. We used Leapfrog to show that the parsers accept the same packets, even though they do so in different ways.

**Variable-Length Formats.** Handling formats with variable lengths, such as type-length-value (TLV) encodings, is a common challenge in protocol parsing, because the amount of data parsed in each state depends on a previously-parsed values. We implemented a parser for Internet Protocol options [8], a common variable-length networking format. Our parser handles up to two generic options, with data-dependent lengths that range from 0 bytes to 6 bytes. We also implemented a custom parser for the Timestamp option, in which a specialized parser extracts the fields specific to its format. Again, we used Leapfrog to show that the parsers accept the same packets, even though the header formats are variable and they do so in different ways.

**Header Initialization.** A common error in P4 programs is reading from uninitialized headers. In parsers, this can happen when several paths converge on a common state,

and the programmer has forgotten to write to a given header on one or more of the paths. For example, VLAN tags [58] are an optional 4-byte format that can appear at the end of an Ethernet frame. If the VLAN tag is present, its value can be used to influence routing behavior. However, a common bug is to accidentally branch on an uninitialized VLAN tag when it was not present in the packet. To fix this bug, one can assign a default value to missing VLAN tags. We implemented a parser for Ethernet, optional VLAN, IP, and UDP, that either parses a VLAN tag or fills it with a default value if it is missing (Listing A.1 of the appendix). We used Leapfrog to check that the set of accepted packets is independent of the initial store. This check succeeds, so we conclude that the parser *not* depend on uninitialized headers.

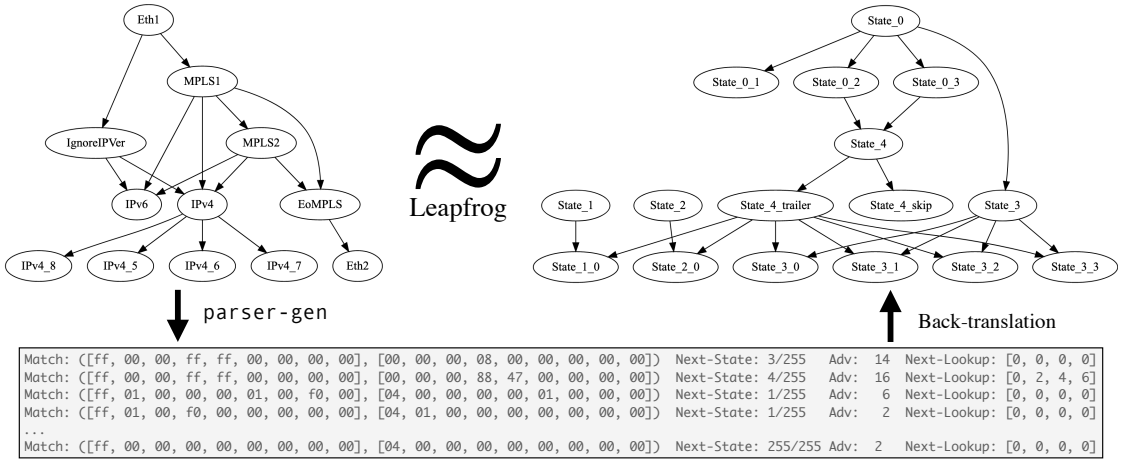
**Speculative Extraction.** Many high-performance protocol parsers *speculatively extract* packet data and then make control-flow decisions based off the contents of that data. We implemented the example from Figure 4.1 with MPLS followed by UDP, in which the body of the optimized MPLS loop speculatively extracts two MPLS headers. If the first of these indicates the end of the header, then the parser has overshot the MPLS header, and the remaining data must be reinterpreted as a UDP packet. We used Leapfrog to verify that these parsers accept the same packets.

**External Filtering.** Another common idiom is to implement a lenient parser that accepts well-formed and malformed packets, and then compensate with an external filter (e.g., by dropping packets later). Recall that the last two bytes of an Ethernet header are a type field that determines the header that follows—e.g., IPv4, IPv6, or something else. We implemented two parsers: a lenient parser that assumes the input packet is IPv6 if it is not IPv4, and a strict parser that explicitly checks the Ethernet type field and rejects other types of packets. We modeled an external filter for the lenient parser by picking an initial relation that not only requires the states to be equally

```
parse_ip {
  extract(ip, 64);
  select(ip[40:43]) {
    (0001) ⇒ parse_udp
    (0000) ⇒ parse_tcp
  }
}
parse_udp {
  extract(udp, 32);
  goto accept
}
parse_tcp {
  extract(tcp, 64);
  goto accept
}
```

```
parse_combined {
  extract(ip, 64);
  extract(pref, 32)
  select(ip[40:43]) {
    (0001) ⇒ accept
    (0000) ⇒ parse_suff
  }
}
parse_suff {
  extract(suff, 32);
  goto accept
}
```

Figure 4.7: Reference and combined parsers for a stylized IP and TCP/UDP protocol.




```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00])  Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00])  Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00])  Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00])  Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00])  Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```

Figure 4.8: The Edge stack case study. The original parser (left) is compiled to a table (below, most entries elided), which we translate back into a parser (right) and prove equivalent to the original.

accepting, but to also terminate with a store where the Ethernet type is IPv4 or IPv6. We then computed a bisimulation modulo this initial relation and prove that the two parsers are equivalent. This case study shows that Leapfrog can do more than just relate the sets of accepted packets: it can also relate the values in the stores.

**Relational Verification.** Leapfrog can also verify other useful relational properties of parsers. For instance, consider two parsers that extract data into differently named

headers (e.g., the example from Section 4.2), or even with different fragments of the input packet scattered across the store. Leapfrog can be used to phrase and verify relations between parser stores. To demonstrate this, we verified that when the lenient and strict parsers from the previous case both accept some packet, there is a correspondence between the values in their stores. We picked an initial relation that requires the values for headers on the left to correspond to the values for headers on the right, provided both configurations are accepting and used Leapfrog to establish a pre-bisimulation. Compared to language equivalence, these applications do not have as much metatheory developed in Coq, where there is a lemma connecting an appropriately configured pre-bisimulation to language equivalence. However, we believe our technique is sound and could be justified in Coq.

Separately from these six tasks, we used Leapfrog to compare parsers that did *not* accept the same packets, such as the two parsers of the *external filtering* task. This was done as a sanity check to see if (1) the proof script still terminated and (2) it did not erroneously claim to prove equivalence. A failure would indicate a bug in our pen-and-paper analysis of the algorithm, or our trusted codebase. Fortunately, Leapfrog acts as expected, by reaching the end of the main loop, then failing when trying to apply the `Close` step.

### 4.7.2 Applicability

To evaluate Leapfrog's applicability to real-world parsers, we encoded the benchmarks used by the developers of the `parser-gen` tool [44]. It provides parsers for *four* different scenarios: (1) Edge, for a gateway router, (2) Service Provider, for a core router, (3) Datacenter, for a top-of-rack switch in a cloud, (4) and Enterprise, for a router in a campus or company network. Each of these parsers supports a different set of

protocols depending on its intended use.[6] We translated each of these parsers into a corresponding P4A parser and used Leapfrog to perform a *self-comparison* check—i.e., we verified that each parser is equivalent to itself.

Next, we used Leapfrog to perform translation validation. The `parser-gen` framework also comes with a compiler that takes a parse graph (analogous to a P4A) and compiles it to an efficient hardware representation. The compiler models constraints at the hardware level (e.g., limiting the number of bits that can be extracted or branched on in each state) and incorporates sophisticated optimizations to make the best use of limited resources (e.g., splitting and merging states).

We ran the `parser-gen` compiler on the parser for the Edge router, which generated a hardware-level representation with states, instructions, and transitions encoded in a table—see Figure 4.8. We then wrote a script to translate the table representation back into a P4 automaton.[7] Finally, we used Leapfrog to check the equivalence of the two parsers.

We were able to prove that the `parser-gen` compiler preserves the semantics of the original Edge P4A automata. Hence, Leapfrog was able to validate a third-party compiler's output on its own benchmark program. Note that we designed Leapfrog before we had experience using `parser-gen`.

[6] We did not consider one of the parsers discussed in the `parser-gen` paper, Big-Union, which models the combined features from all four scenarios. Unlike the others, Big-Union does not model a typical scenario but is primarily intended for bounding hardware requirements.

[7] While the two languages are similar, the parser-gen hardware representation is different enough from P4A (mainly due to unproductive states and speculative lookahead transitions) to make the reverse translation fuzzy. Of all of the parser-gen benchmarks, we found that Edge's hardware table was the closest to P4A and required the least amount of manual repair. This technique could in principle be adapted to other parser-gen benchmarks; while they are a bit larger and could stress Leapfrog's scaling, the main challenge is a robust translation from hardware tables to P4A.

### 4.7.3 Discussion

Overall we find that Leapfrog can be applied to a diverse set of practical scenarios. In the rest of this section, we discuss some of our experiences using the tool, including its limitations and directions for future work.

**Automation** In the early stages of this work, we derived and validated the relevant bisimulations without automated tactics. This turned out to be a significant proof burden—e.g., our manual equivalence proof for the State Rearrangement case study took two weeks of work. In contrast, the push-button Leapfrog proof takes only six seconds on a laptop. Although Leapfrog could be adapted to be more interactive, letting the user apply `Skip` or `Extend` and prove the required entailment, we believe that its power lies in the convenience offered by delegating goals to an SMT solver.

Leapfrog is particularly useful in situations where it is difficult to see whether two parsers are equivalent, such as in the translation validation experiment. While we spent a few days trying to prove the translation validation parsers equivalent on pen-and-paper, we were unsure that they were actually equivalent until the Leapfrog proof succeeded.

**SMT Solver Performance** SMT solvers have unpredictable performance. We used Z3 for the queries in most of our benchmarks, but sometimes needed to switch to CVC4. Overall we found that all of the queries were solved in at most 10 seconds, with 99% taking at most 5 seconds. It was easy to switch between SMT solvers because we targeted a well-supported subset of the SMT-LIB query format (namely the theory of bitvectors).

**Overall Performance** Like any verification tool, Leapfrog has limitations. Scaling to large parsers is challenging due to the combinatorial explosion of configurations. All

of the smaller experiments (up to around 10 states) were interactive on stock hardware, finishing in ≤ 5 minutes and ≤ 16 GB of memory. Most took several minutes. For larger experiments, the larger state space lead to significantly higher memory demands. Coq needed 400 GB of RAM to verify the largest Applicability study (Datacenter) and ran out of memory on the Service Provider study. Although this is a lot, it's unsurprising because the concrete state space for the Applicability study would have around $2^{242}$ elements.

The optimizations discussed in Section 4.5 had a significant impact. Specifically, our smallest State Rearrangement benchmark went from 30 seconds and 1.7 GB of memory to 42 minutes and 36 GB of memory when leaps were disabled; it did not finish without reachable state pruning.

**Future Work** One way to improve the scalability of Leapfrog in the future could be to investigate compositional reasoning techniques. Such techniques could facilitate divide-and-conquer strategies, allowing Leapfrog to be applied to larger parsers than our current implementation supports.

Another possibility is to vary the underlying algorithm. One could imagine a symbolic treatment of Hopcroft and Karp's algorithm [56], which approximates a suitable bisimulation from below, or Paige and Tarjan's *partition refinement* algorithm [86], which represents the current approximation of the largest bisimulation in terms of its equivalence classes. For the latter, one would need to estimate the number of configurations in a symbolically represented equivalence class to choose the next block to split.

The bulk of Leapfrog's memory usage is occupied by the proof object generated by our $L_{tac}$ script. Alternatively, one could implement the same algorithm in Gallina, axiomatizing the decision procedure, and extract it to OCaml. While such an approach is likely to be more efficient, it would also undermine our goal of producing a proof

object that is reusable in a larger verification effort.

P4As are an abstraction of P4 parsers. For one thing, they do not incorporate externs, which are architecture-specific extensions that support, for instance, checksum algorithms or persistent state. In addition, P4 parsers support arrays (in the form of header stacks), subparser calls, and parser lookahead, all of which are not part of our definition of P4 automata. More work is necessary to see whether P4As can be extended to support or simulate these features.

In the future, we would like to use Leapfrog's equivalence checks to systematically perform translation validation on other networking stacks. For example, one could imagine writing a library of reference implementations for protocols defined in RFCs, and checking that real-world implementations conform to those standards.

## 4.8 Acknowledgments

We thank Glen Gibb for help understanding and using his `parser-gen` framework. We received helpful feedback on the writing from Glen Gibb, James R. Wilcox, Bill Harris, and members of the Cornell programming languages group; we thank them for their feedback.

R. Doenges and N. Foster were supported in part by the National Science Foundation under grant FMiTF-1918396, DARPA under contract HR0011-20-C-0107, and gifts from Fujitsu, Google, InfoSys, and Keysight.

T. Kappé was partially supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101027412 (VERLAN). J. Sarracino and G. Morrisett were supported by DARPA contract HR0011-19-C-0073.

CHAPTER 5

# EQUIVALENCE OF TABLE PROGRAMS

## 5.1 Introduction

In this chapter we describe a solver-aided translation validation technique for optimizations on P4 controls, that is, loop-free programs that use tables, and its realization in a tool TTV. We focus in particular on validating the results of table-specific optimizations which arise in data plane compilers out of necessity.

Existing verified translation validators target particular optimization phases and use a sound, complete, and simple validation algorithm that is implemented and verified entirely by hand. The CompCert register allocator [70] is the paradigmatic example of such a validator. Expanding the purview of translation validation from a particular optimization phase to an entire compiler runs up against foundational obstacles like the undecidability of program equivalence for Turing-complete languages.

In the case of P4, program equivalence is actually decidable, so there is hope for a translation validator that validates entire compilations. In fact, such a validator has already been built by Ruffy et al. [93]. The Gauntlet translation validator found many bugs in the open source P4C compiler toolchain. It is implemented in Python and uses the Z3 solver to check program equivalence, so it does not produce certificates and relies on an unverified symbolic execution engine to generate solver queries. It also fails to validate certain optimizations involving rearrangements of tables.

Table optimizations are not required in intermediate phases of P4 compilers but take on a special importance in back ends that target hardware. Dedicated hardware targets for P4 are resource constrained but offer predictable performance within their hard resource bounds. For instance, tables are typically implemented in hardware by content-adressable memories (CAMs) such as TCAMS (ternary CAMs), which are

smaller than RAMs. All the rules of a large routing table may not fit into a single TCAM, so the table may have to be split across several TCAMs in hardware in order to fit. Analogously, tables may be merged in order to keep the total number of TCAMs occupied from exceeding the target architecture's limits. All this allocation has to be handled by the compiler at compile time, and if the compiler cannot fit the program into the target's resource bounds it fails.

Table optimizations can challenge our understanding of what it means for two programs to be equivalent. Compare the control in Figure 5.1 with the control in Figure 5.2. Given a set of addresses to monitor, these programs can be configured to count the number of packets intended for those addresses. This is accomplished with configurable tables which have the same key and the same actions in both controls. The control in Figure 5.1 has one table while Figure 5.2 has two that run in sequence.

Any configuration of the single table control (e.g., the rules in Table 5.1), can be translated into a configuration for the two table control (e.g., the rules in Table 5.2) that behaves the same. The converse is not true. The configuration in Table 5.3 can cause the two table program to increment the counter *twice*, but all configurations of the single table control (Figure 5.1) cause it to increment the counter at most *once*.

Since the two controls do not exhibit the same behaviors, they must not be equivalent. Gauntlet [93] actually uses this notion of equivalence (or at least something close to it). In particular, it rejects the table-splitting optimization.

This criterion is too restrictive for a complete P4 compiler because, as argued above, table rearrangements (including table splitting) are necessary in order to put programs within the resource bounds of hardware targets. With this in mind, we develop a translation validation technique that checks *control plane simulation* instead of program equivalence.

A program $p$ *control plane simulates* a program $q$ if all control plane configurations

of $q$ can be related to control plane configurations of $p$ in a behavior-preserving manner. In our example, there is a simulation that relates the configurations in Table 5.1 and Table 5.2, but the simulation is not right-total (e.g., there are some configurations of the two table control which do not correspond to configurations of the single table control, as discussed above).

In a real P4 system, control plane simulations are actually enforced by a control plane shim interposed between the optimized program and the control plane because the compiled program still needs to match the original program's control plane interface. The shim accomplishes this by translating rules meant for the original program into rules meant for the optimized program.

We implement this approach in a prototype validator called TTV, which validates optimizations of programs in a core calculus similar to P4 controls. Following our approach in Chapter 4, we implement TTV in a mix of Coq and OCaml and rely on an external SMT solver to dispatch verification conditions (VCs). This architecture allows the VC generation phases to be removed from the TCB by verification in Coq, but allows us to take advantage of the well-engineered, fast, and mature decision procedures in off-the-shelf SMT solvers. As input, TTV takes a source and target program and a logical specification of a control plane simulation. It outputs a boolean stating whether the simulation relates the two programs or not. In the remainder of this chapter we discuss some of the technical challenges that arose in building TTV, including the design of syntax for control plane simulations, the new solver interface required to make the syntax work, and the design of a relational verifier. Then we give a formal presentation of our languages, logics, and VC generation and describe the implementation.

```
control C(inout headers h, inout bit<8> counter) {
  action NoAction() { }
  action count() {
    counter = counter + 1;
  }
  table t {
    key = { h.dst: exact };
    actions = { NoAction(); count() };
    default_action = NoAction();
  }
  apply {
    t.apply();
  }
}
```

Figure 5.1: Example: P4 control with one table.

```
control C(inout headers h, inout bit<8> counter) {
  action NoAction() { }
  action count() {
    counter = counter + 1;
  }
  table t1 {
    key = { h.dst: exact };
    actions = { NoAction(); count() };
    default_action = NoAction();
  }
  table t2 {
    key = { h.dst: exact };
    actions = { NoAction(); count() };
    default_action = NoAction();
  }
  apply {
    t1.apply();
    t2.apply();
  }
}
```

Figure 5.2: Example: P4 control with two tables.

| Table | Key | Action |
|---|---|---|
| t | 192.168.1.10 | count |
| t | 192.168.1.11 | count |
| t | 192.168.1.12 | count |

Table 5.1: Control plane state for the control in Figure 5.1.

| Table | Key | Action |
|---|---|---|
| t1 | 192.168.1.10 | count |
| t2 | 192.168.1.11 | count |
| t2 | 192.168.1.12 | count |

Table 5.2: Control plane state for the control in Figure 5.2.

| Table | Key | Action |
|---|---|---|
| t1 | 192.168.1.10 | count |
| t1 | 192.168.1.11 | count |
| t1 | 192.168.1.12 | count |
| t2 | 192.168.1.10 | count |
| t2 | 192.168.1.11 | count |
| t2 | 192.168.1.12 | count |

Table 5.3: Control plane state with duplicate rules for the control in Figure 5.2.

### 5.1.1 The Relational Verification Problem

Traditional program verification focuses on sequential properties, which are properties of individual programs. Program equivalence, the chief concern of a translation validator, is not a sequential property because it relates the behavior of two programs. The appropriate theoretical setting for translation validation is relational verification, which is studied in logical form with Relational Hoare Logic (RHL).

$$\vdash \{R\}\ p \sim q\ \{S\}$$

RHL assertions work like assertions in ordinary (i.e., sequential) Hoare logic, but compare two executions of two programs instead of describing one execution of one program. Here $p$ and $q$ are programs with disjoint sets of variables and $R$ and $S$ are logical formulas over the variables of $p$ and $q$. The judgment holds when given initial configurations $\sigma_p$ and $\sigma_q$ for $p$ and $q$ such that $R$ is true, running $p$ in $\sigma_p$ and running $q$ in $\sigma_q$ always produces final configurations $\sigma'_p$ and $\sigma'_q$ satisfying $S$.

Like Hoare Logic, RHL enjoys a relative completeness property. The proof reduces RHL assertion validity to ordinary Hoare Logic assertion validity by combining the two programs into one with the so-called “product construction.”

### 5.1.2 The Trustworthy Solver Problem

Building a verifier based upon the relative completeness of a program logic means that the soundness of the verifier depends upon the correctness of its lowering from its rich program logic to the pure base logic and on the procedure for checking the base logic assertion. In reality this base logic is usually a theory in SMT-LIB or another concrete format, checked by an SMT solver, and the lowering may not take the most obviously sound approach to encoding things in the base logic, than is, it may make optimizations. The soundness of solvers has been studied and many solvers are able to produce checkable certificates for their results. "Can I trust the solver?" is a less interesting question in my view than "Can I trust the verifier with a solver?" Optimizations and performance hacks in verifiers are one-off bits of ingenuity and trusting their results before trusting the solver is a misapprehension of how risk is distributed in solver-aided systems.

In this light, a lowering from a program logic to a base logic looks less like a simple translation and more like an optimizing compiler, and all the risks of compiler development recur here with a logical twist. Rather than endangering the correct operation of the source program, a bad lowering endangers the validity of the source assertion. However, these lowerings are generally not verified.

In Leapfrog (Chapter 4) we verified a lowering from a language of bisimulations to a first order logic over a fixed signature. Encoding tables as function symbols goes beyond what this approach can account for because it requires making part of the signature of the base logic depend upon the names and types of the tables that are present in the source program. The main obstacle to extending the Leapfrog approach is a practical one: the Leapfrog first-order logic is defined in Coq as an inductive type family indexed by signatures. Programming a translation that produces a different signature depending on its input is a kind of complex dependently-typed

programming that is difficult to accomplish in Coq and inefficient once completed.

## 5.2 GGCL Verification Framework

In this section we momentarily put aside P4-related concerns to formalize GGCL, the generic guarded command language. Having defined the programming language we then describe a method of relational verification and show how to instantiate GGCL to approximate P4 table programs.

GGCL is an imperative language in the tradition of Dijkstra's GCL which has been parameterized, ISWIM-style, over a first-order signature and an intended interpretation for the signature. This hides enough domain-specific detail to make it simple to work with but is flexible enough to capture all the features of P4 we will be concerned with.

### 5.2.1 First-Order Signatures

A signature $\Sigma$ collects sorts $X \in \Sigma$, function symbols $f \in \Sigma$, and relation symbols $r \in \Sigma$. Function symbols are equipped with arities $X_1 \times \cdots \times X_{n-1} \to X_n$. Relation symbols are equipped with arities $X_1 \times \cdots \times X_n \to \mathbb{B}$.

Signatures may be infinite. For instance, a signature with a bitvector type $\mathsf{bv}k$ for every natural number $k$ will have countably infinite sets of sorts, function symbols, and relation symbols. This means that programs manipulating signatures must rely on some kind of finitary representation, a problem for our encoding, but we will defer the resolution of this problem to the implementation section.

### 5.2.2 First-Order Logic

Fixing a signature $\Sigma$, the syntax of first-order logic (FOL) is as follows. We omit the definitions of well-sortedness judgments $\Gamma \vdash e : X$ and $\Gamma \vdash \varphi : \mathbb{B}$, which check that terms respect the arities of function and relation symbols.

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
& | & f(e, \ldots, e) & \text{function} \\
\varphi & ::= & \top & \text{true} \\
& | & \bot & \text{false} \\
& | & \varphi \wedge \varphi & \text{conjunction} \\
& | & \varphi \vee \varphi & \text{disjunction} \\
& | & \varphi \Rightarrow \varphi & \text{implication} \\
& | & \neg \varphi & \text{negation} \\
& | & e = e & \text{equality} \\
& | & r(e, \ldots, e) & \text{relation} \\
& | & \forall x : X . \varphi & \text{universal quantification} \\
& | & \exists x : X . \varphi & \text{existential quantification}
\end{array}
$$

In the sequel, sorts $X$ on quantifers may be omitted where they can be inferred and we may use the metavariable $\varphi_{\mathrm{QF}}$ to refer to quantifier-free formulas.

### 5.2.3 The Generic Guarded Command Language (GGCL)

We now introduce the Generic Guarded Command Language or GGCL. Like the logic, it is parameterized over a first-order signature. It is an imperative language with nondeterminism and no loops (to match P4). Note the use of FOL expressions $e$ in variable assignments and the use of FOL (quantifier free) forumulas $\varphi_{QF}$ in assume and assert statements. There is an unsurprising typing judgment $\Gamma \vdash s$ which enforces

well-sortedness with respect to a context $\Gamma$ of sorted variables.

$$
\begin{array}{rcll}
s & ::= & \mathsf{skip} & \text{skip} \\
& | & x := e & \text{assignment} \\
& | & s; s & \text{sequencing} \\
& | & s \,\square\, s & \text{nondeterministic choice} \\
& | & \mathsf{assert}\ \varphi_{QF} & \text{assertion} \\
& | & \mathsf{assume}\ \varphi_{QF} & \text{assumption}
\end{array}
$$

We now give predicate transformer semantics to GGCL in the form of a weakest precondition operator $\mathrm{wp}(s, \varphi)$.

$$
\begin{aligned}
\mathrm{wp}(\mathsf{skip}, \varphi) &= \varphi \\
\mathrm{wp}(x := e, \varphi) &= \varphi[e/x] \\
\mathrm{wp}(s_1; s_2, \varphi) &= \mathrm{wp}(s_1, \mathrm{wp}(s_2, \varphi)) \\
\mathrm{wp}(s_1 \,\square\, s_2, \varphi) &= \mathrm{wp}(s_1, \varphi) \wedge \mathrm{wp}(s_2, \varphi) \\
\mathrm{wp}(\mathsf{assert}\ \psi, \varphi) &= \psi \wedge \varphi \\
\mathrm{wp}(\mathsf{assume}\ \psi, \varphi) &= \psi \Rightarrow \varphi
\end{aligned}
$$

This is sufficient to prove sequential properties of programs, but cannot perform program equivalence checks. For relational properties, we will have to turn to the product construction.

Given two GGCL programs $s_1$ and $s_2$ with distinct free variables, another program $p$ is a *product program* if its input-output behavior on the variables of $s_1$ is identical to that of $s_1$ and likewise for $s_2$. In effect, $p$ captures all the behaviors of $s_1$ alongside all the behaviors of $s_2$. There are many possible product programs, but the simplest is

$$s_1; s_2.$$

Hereafter we refer to this as *the* product program or product construction. Any relational postcondition $R$ of the programs $s_1$ and $s_2$ may be verified by performing the

weakest precondition on the product program, that is, by checking the validity of the precondition

$$\mathrm{wp}(s_1; s_2, R).$$

Some care is required in the handling of free variables. They may need to be renamed both in programs and in specifications in order to prevent clashes in the product construction.

### 5.2.4 Encoding P4 in GGCL

An appropriate instantiation of GGCL can capture the behavior of P4 programs and tables. The key encoding choice here is to treat a table as an uninterpreted function with a parameter for each table key and a return value that indicates the chosen action and its arguments.

We begin by defining necessary P4 sorts: bitvectors and products.

$$\begin{array}{rcll} X & ::= & \mathsf{bit}\, k & \text{bitvectors of width } k \\ & | & X \times X & \text{tuples} \end{array}$$

There are always at least the following function symbols, but we adjoin more symbols to represent tables. So the signature in use depends on the program(s) being analyzed.

$$\begin{array}{rcll} f & ::= & b_1 b_2 \ldots b_k & \text{bitvector literal} \\ & | & \pi_1^{X,Y} & \text{project first tuple component of } X \times Y \\ & | & \pi_2^{X,Y} & \text{project second component} \\ & | & \ominus_k y & \text{unary operators over k-width bitvectors} \\ & | & x \oplus_k y & \text{binary operators over k-width bitvectors} \end{array}$$

Bitvector literals are 0-ary symbols (i.e., constants). Unary and binary operators may be drawn from a collection of symbols including at least signed and unsigned addition,

subtraction, and so on. The tuple projections are sort-indexed to make it so they still have unique arities ($X \times Y \to X$ and $X \times Y \to Y$ respectively).

There are no relation symbols. Recall that we treat equality as a primitive and do not model it as a relation symbol in our logic.

The construction of table symbols is the most interesting part of this exercise. Consider the following P4 table modeled on the tables in Figure 5.1 and Figure 5.2.

```
table t {
  key = { h.dst: exact };
  actions = { NoAction(); count() };
  default_action = NoAction();
}
```

Assume the P4 type of `h.dst` is `bit<8>`. Then the ranked function symbol associated with `t` is

$$t : \mathsf{bit}\,8 \to \mathsf{bit}\,1.$$

The argument to $t$ is meant to be `h.dst` and the return value is an index into the `actions` list.

Thanks to products we can also account for tables which provide an argument to their actions. Consider the following program fragment.

```
action count(bit<8> k) {
  counter = counter + k;
}
table tk {
  key = { h.dst: exact };
  actions = { NoAction(); count() };
  default_action = NoAction();
}
```

The appropriate function symbol for `tk` is

$$tk : \mathsf{bit}\,8 \to \mathsf{bit}\,1 \times \mathsf{bit}\,8,$$

where the second component of the returned tuple is only meaningful when the `count` action is chosen.

### 5.2.5 Encoding Control Plane Simulations

There is a clear relationship between interpretations of table function symbols (that is, mathematical functions of appropriate type) and P4 control plane configurations. This allows us to encode the control plane simulations necessary to justify table optimizations as closed formulas $\varphi$ over table function symbols.

For example, if the function symbols for `t1` and `t2` are $t_1$ and $t_2$ respectively, we can justify our splitting optimization with the following control plane simulation. It has two conjuncts. The first says $t_1$ and $t_2$ cannot both choose to count the same packet and the second says that $t$ should match either $t_1$ or $t_2$.

$$\forall x. \neg(t_1(x) = 1 \wedge t_2(x) = 1) \wedge (t(x) = t_1(x) \vee t(x) = t_2(x)) \tag{5.1}$$

### 5.2.6 Verifying The Example

In this section we return to the programs in Figure 5.1 and Figure 5.2 to show how to prove them equivalent using an appropriately instantiated GGCL.

The table symbols in our signature shall be

$$t, t_1, t_2 : \mathsf{bit}\,8 \to \mathsf{bit}\,1.$$

The GGCL programs in Figure 5.3 and Figure 5.4 corresponds to the P4 programs in Figure 5.1 and Figure 5.2, respectively. A product program is shown in Figure 5.5. Note that the variables of the second program have been primed to avoid name clashes.

$$t_{res} := t(dst); \big((\text{assume } t_{res} = 1; counter := counter + 1) \,\square\, (\text{assume } t_{res} = 0; \text{skip})\big)$$

Figure 5.3: Example: GGCL program with one table.

$$\begin{aligned} t_{1,res} &:= t_1(dst); \big((\text{assume } t_{1,res} = 1; counter := counter + 1) \,\square\, (\text{assume } t_{1,res} = 0; \text{skip})\big); \\ t_{2,res} &:= t_2(dst); \big((\text{assume } t_{2,res} = 1; counter := counter + 1) \,\square\, (\text{assume } t_{2,res} = 0; \text{skip})\big) \end{aligned}$$

Figure 5.4: Example: GGCL program with two tables.

The relational postcondition we would like to check is

$$counter = counter'.$$

The relational precondition is

$$counter = counter' \wedge dst = dst'.$$

Let $s$ stand for the product program in Figure 5.5 and let $\psi$ be the control plane simulation of Equation (5.1). The complete verification condition is then

$$\psi \wedge counter = counter' \implies \text{wp}(s, counter = counter').$$

## 5.3 Implementation

We implement TTV as a solver-aided program in a mixture of Coq and OCaml. The Coq library defines a first order logic, GGCL, and a P4-specific signature. It includes a weakest precondition generator translating GGCL to FOL verification conditions and an implementation of the product construction. The OCaml code handles pretty-printing the FOL syntax to SMT-LIB and sending it to SMT solvers.

$$\begin{aligned} t_{res} &:= t(dst); \big((\text{assume } t_{res} = 1; counter := counter + 1) \,\square\, (\text{assume } t_{res} = 0; \text{skip})\big); \\ t'_{1,res} &:= t_1(dst'); \big((\text{assume } t'_{1,res} = 1; counter' := counter' + 1) \,\square\, (\text{assume } t'_{1,res} = 0; \text{skip})\big); \\ t'_{2,res} &:= t_2(dst'); \big((\text{assume } t'_{2,res} = 1; counter' := counter' + 1) \,\square\, (\text{assume } t'_{2,res} = 0; \text{skip})\big) \end{aligned}$$

Figure 5.5: Example: GGCL product program.

### 5.3.1 Coq Library

The Coq library defines syntax for GGCL and first order logic. It includes a weakest precondition transformer for GGCL programs and a generic product construction. These Coq definitions are more or less direct transcriptions of what is done on paper in this chapter. All of these constructs are parameterized over a signature, but the encoding of the signature is *not* done as shown in this chapter.

Instead, the signature in the implementation is separated into infinite and finite parts. The infinite part may have infinitely many function, sort, and relation symbols. These sets are represented by Coq types. The finite part may have only finitely many of each kind of symbol and the sorts are represented by finite maps.

This setup complicates the parameterization of operations, but it has important benefits. First, it reduces the notational overhead of GGCL programming. Each P4 program uses a slightly different signature because each P4 program uses a slightly different collection of named tables. If signatures were one type, this would mean the GGCL terms for different P4 programs would have different types (and type arguments). Also, for pretty-printing, it is useful to have an actual data structure of uninterpreted function symbols so that they can be printed as SMT-LIB declarations.

### 5.3.2 SMT Solver Integration

The integration with SMT solvers is implemented in OCaml. The Coq extraction mechanism produces reasonable OCaml from the TTV GGCL and FOL syntaxes. This is a marked improvement on the highly dependent syntax of Leapfrog, which is not typeable by OCaml without the insertion of unsafe dynamic casts.

To check VCs, the OCaml pretty-printer prints the signature of a VC as a list of SMT-LIB directives and then prints the VC itself as a negated SMT-LIB assertion. The solver

checks the satisfiability of this negated assertion, which is equivalent to checking the validity of the original assertion.

### 5.3.3 Trusted Computing Base

The verification of the reductions in TTV is not complete as of this writing. As such, its TCB is like that of any solver-aided tool. We describe here the desired TCB, which we plan to reach with further proof engineering.

Our theorems, once proved in Coq, will mean the reduction from pairs of GGCL programs to FOL formulas need not be trusted provided Coq's proof kernel and its extraction mechanism are trusted. The pretty-printer must be trusted and the solver must be trusted. In particular, the pretty printer should respect our semantics of FOL and the semantics of SMT-LIB as defined in the SMT-LIB specification.

## 5.4 Conclusions and Future Work

We have described TTV, an in-progress verified translation validator. It uses a carefully designed generic programming language and off-the-shelf SMT solvers to prove relational properties of table programs without dragging a complex logical translation into the trusted computing base of the system. This improves upon the TCB of traditional solver-aided translation validators, which are very powerful tools but generally exchange a complex but heavily tested compiler for a similarly complex but less mature translation to SMT.

In future work we hope to complete the verification of our translation in Coq and evaluate TTV by using it on existing P4 compilers. We are particularly interested in finding more complex table optimizations to study.

CHAPTER 6

# RELATED WORK

## 6.1 Related Work (Petr4)

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 [78]. More recently, a number of prominent efforts have developed semantics for languages as complex and diverse as JavaScript [49, 87], WebAssembly [51], C [71], x86-TSO [96], and the POSIX shell [46]. Like PETR4, these efforts build on decades of foundational work in semantics [95, 89, 64] and semantics engineering [97]. Recent work by Ruffy et al. has profitably combined fuzzing and translation validation to find numerous bugs in P4C [93]. 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 [13] 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 [48]. Another line of recent work has focused on eBPF, the packet-processing framework supported in the Linux kernel. The JitK compiler [110] 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 [42].

**Network Verification** As mentioned in Section 3.1, there is a growing body of work focused on data plane and control plane verification, including Header Space Analysis [67], Anteater [75], NetKAT [3], Batfish [40], ARC [43], and Minesweeper [11], to name a few. Other tools have applied techniques such as predicate transformers [73], symbolic execution [101, 84], or translation into another language, such as Datalog [77], 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 [68]. 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.

## 6.2 Related work (Leapfrog)

**Automata Equivalence Checking.** Our algorithm is a variation on Moore's classical algorithm to decide all-pairs language equivalence in a DFA [79]. Moore's approach was later improved upon by *partition refinement* [55, 65, 86]. We deviate from these classical procedures in two key aspects.

First, instead of using concrete data structures we use symbolic ones. This idea goes back to Coudert et al. [26], and has since been widely applied [18, 19, 21, 20, 33]. These algorithms use Binary Decision Diagrams (BDDs) as their symbolic representation. Other authors favored a logical representation, combined with decision procedures for the logic [52, 39]. Dehnert et al. [31] makes use of an SMT solver to decide questions about the logical representations.

Second, instead of maintaining a list of equivalence classes, we maintain a representation of an equivalence relation. The earliest instance of this we have been able to track down is due to Bouali and De Simone [20]. Mumme and Ciardo observed that such an approach is particularly beneficial when there tend to be a large number of equivalence classes [80].

Algorithms based on *bisimulation up to congruence* [15, 29] are similar in the sense that they mitigate state space explosion—in this case, as a result of determinization. They exploit the internal structure of the expanded state space to terminate early, something that inspired us propose the notion of a bisimulation with leaps.

**Network Verification.** The `p4v` verifier [73], the verifier of Neves et al. [82], and Aquila [106] are push-button verifiers for functional properties of P4 programs, including P4 parsers. They work by translating to a verification IR (either guarded command language [34] or simple C) and then analyzing the IR. None of these tools produce proofs, and their translations are not proved sound with respect to a reusable se-

mantics of P4. Moreover, these tools verify *functional* specifications about a single P4 program. Our work is complementary because by contrast, our tool produces *relational* proofs grounded in a reusable Coq semantics for two P4 automata. Aquila includes a self-validation system for finding semantic bugs in the verifier. Defining our semantics in Coq allowed us to foundationally *prove* the absence of semantic bugs, so while Leapfrog does not need self-validation, it could be an oracle for validating other tools.

The Gauntlet translation validator checks program equivalence for P4 programs without parsers or externs. We see this work as complementary to Leapfrog, which focuses on parser equivalence. Outside the parser, P4 programs have loop-free control flow, complex data structures, and rich semantic actions. Inside the parser, P4 programs have loops, simpler data structures, and simpler semantic actions. Consequently, parser verification is concerned with control flow more than anything else, making it a different kind of verification problem than verification for the rest of a P4 program.

**Automatic Foundational Verification** SpaceSearch [112] exposes a high-level solver interface to search large state spaces. In contrast, our solver interface is lower level, and our tool avoids extraction to produce a Coq certificate.

CreLLVM [66] instruments LLVM to produce translation validation proofs in a relational Hoare logic, resulting in a compact TCB and reusable Coq proof certificate. Leapfrog has a similar TCB, but completeness (Theorem 4.4.5) means it does not require proof hints.

The Narcissus [32] and EverParse [92] tools synthesize correct parsers and serializers from high level descriptions of packet formats using verified parser combinator libraries. Synthesis and equivalence are related but distinct problems, and our tool is complementary to synthesis tools. For instance, a P4 parser generated by a parser

synthesizer like EverParse might be further optimized by a P4 compiler to run on hardware. Leapfrog could validate the results of compilation, preserving the guarantee offered by the synthesizer.

GPaco [57, 113] is a framework for modular coinductive reasoning in Coq, which supports "up-to" bisimilarity techniques. It is designed for interactive use and focuses on automating low-level proof steps. GPaco may be useful for generalizing our mechanized metatheory for leaps.

APPENDIX A

**LEAPFROG PROOFS AND LISTINGS**

## A.1 Proofs omitted from Section 4.4

**Lemma 4.4.1.** *For formulas $\phi$, the following are equivalent:*

1. *There exists a symbolic bisimulation $\psi$ such that $\phi \vDash \psi$.*
2. *There exists a bisimulation $R$ such that $[\![\phi]\!]_{\mathscr{L}} \subseteq R$.*
3. *If $c_1 \; [\![\phi]\!]_{\mathscr{L}} \; c_2$, then $L(c_1) = L(c_2)$.*

*Proof.* (1) implies (2) by definition of symbolic bisimulation. The proof that (2) implies (3) is a standard argument from automata theory, showing that for all $c_1 \; R \; c_2$ and $w \in \{0,1\}^*$, it holds that $w \in L(c_1)$ if and only if $w \in L(c_2)$, by induction on $w$. Finally, to see that (3) implies (1), we construct $\psi$ to relate all configurations with equivalent languages, i.e.,

$$\psi := \bigvee_{L(c_1)=L(c_2)} \phi^{<}(c_1) \wedge \phi^{>}(c_2) \quad \text{where for } \leqslant\!\!\in \{<,>\},$$

$$\phi^{\leqslant}(q,s,w) := q^{\leqslant} \wedge \bigwedge_{h \in H} h^{\leqslant} = s(h) \wedge \mathrm{buf}^{\leqslant} = w$$

Clearly, $[\![\phi^{<}(c_1) \wedge \phi^{>}(c_2)]\!]_{\mathscr{L}} = \{\langle c_1, c_2 \rangle\}$ for all $c_1, c_2 \in C$, and hence $c_1 \; [\![\psi]\!]_{\mathscr{L}} \; c_2$ if and only if $L(c_1) = L(c_2)$. This makes $[\![\psi]\!]_{\mathscr{L}}$ a bisimulation: configurations with the same language are equally accepting, and step to configurations with the same language; hence, $\psi$ is a symbolic bisimulation. Since states related by $[\![\phi]\!]_{\mathscr{L}}$ agree on their languages, $\phi \vDash \psi$. □

**Theorem 4.4.5.** *Algorithm 1 is correct.*

*Proof.* Note that each iteration either the relation $[\![\bigwedge R]\!]_{\mathscr{L}}$ shrinks (if we enter the **if**-block), or $[\![\bigwedge R]\!]_{\mathscr{L}}$ stays the same but $T$ shrinks. Thus, the main loop must terminate.

For partial correctness, we note that the loop maintains the following three loop invariants:

1. If $c_1 \, [\![ \bigwedge R \wedge \bigwedge T ]\!]_{\mathscr{L}} \, c_2$, then for all $b \in \{0,1\}$ it holds that $\delta(c_1, b) \, [\![ \bigwedge R ]\!]_{\mathscr{L}} \, \delta(c_2, b)$.

   This claim holds trivially before the loop, because $R$ is empty. To see that it is preserved, let $T'$ and $R'$ be the new values of $T$ and $R$. There are two cases.

   - If $\bigwedge R \vDash \psi$ then $R' = R$ and $T' \cup \{\psi\} = T$. Thus,

   $$\begin{aligned} [\![ \textstyle\bigwedge R' \wedge \bigwedge T' ]\!]_{\mathscr{L}} &= [\![ \textstyle\bigwedge R \wedge \bigwedge T' ]\!]_{\mathscr{L}} \\ &= [\![ \psi \wedge \textstyle\bigwedge R \wedge \bigwedge T' ]\!]_{\mathscr{L}} \\ &= [\![ \textstyle\bigwedge R \wedge \bigwedge T ]\!]_{\mathscr{L}} \end{aligned} \tag{A.1}$$

     Since $R = R'$ and the property was true before the loop, it also holds afterwards.

   - Otherwise, $R' = R \cup \{\psi\}$ and $T' = T \cup \mathsf{WP}(\psi)$. Suppose $c_1 \, [\![ \bigwedge R' \wedge \bigwedge T' ]\!]_{\mathscr{L}} \, c_2$; we should show that, for all $b \in \{0,1\}$, we have $\delta(c_1, b) \, [\![ \bigwedge R ]\!]_{\mathscr{L}} \, \delta(c_2, b)$ as well as $\delta(c_1, b) \, [\![ \psi ]\!]_{\mathscr{L}} \, \delta(c_2, b)$. Since $R' \cup T' \subseteq R \cup T$, we have in particular that $c_1 \, [\![ \bigwedge R \wedge \bigwedge T ]\!]_{\mathscr{L}} \, c_2$, and hence $\delta(c_1, b) \, [\![ \bigwedge R ]\!]_{\mathscr{L}} \, \delta(c_2, b)$, because the property is true before the loop.

     Furthermore, since $\mathsf{WP}(\psi) \subseteq R' \cup T'$, we also have $c_1 \, [\![ \chi ]\!]_{\mathscr{L}} \, c_2$ for all $\chi \in \mathsf{WP}(\psi)$. By the precondition about $\mathsf{WP}$, we conclude that $\delta(c_1, b) \, [\![ \psi ]\!]_{\mathscr{L}} \, \delta(c_2, b)$.

2. If $c_1 \, [\![ \bigwedge R \wedge \bigwedge T ]\!]_{\mathscr{L}} \, c_2$, then $c_1 \in F$ iff $c_2 \in F$.

   This property holds by construction before the loop. To see that it is preserved, note that $[\![ \bigwedge R \wedge \bigwedge T ]\!]_{\mathscr{L}}$ stays the same in the iterations where $\bigwedge R \vDash \psi$, and shrinks in all other iterations. Because the claim holds before each iteration, it must also hold afterwards.

3. If $\rho$ is a symbolic bisimulation, then $\rho \vDash \bigwedge R \wedge \bigwedge T$.

   This claim holds before the loop, where $R = \emptyset$ and $T = I$. After all, if $c_1 \, [\![\rho]\!]_{\mathscr{L}} \, c_2$, then $c_1 \in F$ if and only if $c_2 \in F$, and hence $c_1 \, [\![\bigwedge I]\!]_{\mathscr{L}} \, c_2$.

   For preservation, let $T'$ and $R'$ be the new values of $T$ and $R$. There are again two cases.

   - If $\bigwedge R \vDash \psi$, then by the derivation in (A.1), we know that $[\![\bigwedge R' \wedge \bigwedge T']\!]_{\mathscr{L}} = [\![\bigwedge R \wedge \bigwedge T]\!]_{\mathscr{L}}$. Since $\rho \vDash \bigwedge R \wedge \bigwedge T$, it then follows that $\rho \vDash \bigwedge R' \wedge \bigwedge T'$.
   - Otherwise, we have that $R' \cup T' = R \cup T \cup \mathsf{WP}(\psi)$. We should show that if $c_1 \, [\![\rho]\!]_{\mathscr{L}} \, c_2$, then $c_1 \, [\![\chi]\!]_{\mathscr{L}} \, c_2$ for all $\chi \in R' \cup T'$. Because the property holds before the loop, we already know that, under these circumstances, $c_1 \, [\![\chi]\!]_{\mathscr{L}} \, c_2$ for all $\rho \in R \cup T$; thus, it suffices to prove $c_1 \, [\![\chi]\!]_{\mathscr{L}} \, c_2$ for all $\chi \in \mathsf{WP}(\psi)$. By the precondition on $\mathsf{WP}$, it suffices to verify that for all $b \in \{0,1\}$ we have $\delta(c_1, b) \, [\![\psi]\!]_{\mathscr{L}} \, \delta(c_2, b)$. Now, since $\rho$ is a symbolic bisimulation, we already know that for all $b \in \{0,1\}$ we have $\delta(c_1, b) \, [\![\rho]\!]_{\mathscr{L}} \, \delta(c_2, b)$. Lastly, since the claim held before the loop, and $\psi \in T$, it then follows that $\delta(c_1, b) \, [\![\psi]\!]_{\mathscr{L}} \, \delta(c_2, b)$.

When the loop terminates, all of these conditions are true, and we also have that $T = \emptyset$. The first two conditions then tell us that $\bigwedge R$ is a symbolic bisimulation; moreover, the second condition says that if $\rho$ is a symbolic bisimulation, then $\rho \vDash \bigwedge R$. It then follows that $\bigwedge R$ is in fact the *weakest* symbolic bisimulation. This allows us to wrap up the correctness argument as follows:

- If the algorithm returns **true**, we know that $\phi$ entails a symbolic bisimulation, which by Lemma 4.4.1 tells us that $c_1 \, [\![\phi]\!]_{\mathscr{L}} \, c_2$ implies $L(c_1) = L(c_2)$.
- Conversely, if $c_1 \, [\![\phi]\!]_{\mathscr{L}} \, c_2$ implies $L(c_1) = L(c_2)$, then by Lemma 4.4.1 there exists a symbolic bisimulation $\psi$ such that $\phi \vDash \psi$. Since $\bigwedge R$ is the *weakest* symbolic

bisimulation, we also know that $\psi \vDash \bigwedge R$, and thus $\phi \vDash \bigwedge R$, meaning the algorithm returns **true**. □

## A.2 Proofs omitted from Section 4.5

**Theorem 4.5.2.** *Let $\phi$ be a formula. Algorithm 1 remains correct for this $\phi$ if we set $I$ to the smallest set satisfying the rule*

$$\frac{t_1 \ \mathsf{reach}_\phi \ t_2 \qquad t_1 = t_{\mathsf{accept}} \iff t_2 \neq t_{\mathsf{accept}}}{[t_1^< \wedge t_2^> \implies \bot] \in I}$$

*and for each template-guarded formula $t_1^< \wedge t_2^> \implies \psi$ we set $\mathsf{WP}(t_1^< \wedge t_2^> \implies \psi)$ to the smallest set satisfying the rule*

$$\frac{t_1' \ \mathsf{reach}_\phi \ t_2' \qquad \psi' = \mathsf{WP}^<(\mathsf{WP}^>(\psi', x, t_2', t_2), x, t_1', t_1)}{[t_1'^< \wedge t_2'^> \implies \phi'] \in \mathsf{WP}(t_1^< \wedge t_2^> \implies \phi)}$$

*where $x \in \mathsf{Var}$ is some variable that is fresh for $\psi$.*

*Proof.* The same termination argument still applies. The loop invariants are as follows.

1. If $c_1 \ [\![\bigwedge R \wedge \bigwedge T]\!]_{\mathscr{L}} \ c_2$, then for all $b \in \{0,1\}$ it holds that $\delta(c_1, b) \ [\![\bigwedge R]\!]_{\mathscr{L}} \ \delta(c_2, b)$.

   The proof is completely analogous to the corresponding loop invariant in Theorem 4.4.5.

2. If $c_1 \ [\![\bigwedge R \wedge \bigwedge T]\!]_{\mathscr{L}} \ c_2$ and $\lfloor c_1 \rfloor \ \mathsf{reach}_\phi \ \lfloor c_2 \rfloor$, then $c_1 \in F$ if and only if $c_2 \in F$.

   Note that this invariant is slightly weaker than the corresponding invariant in the proof of Theorem 4.4.5. Here, it suffices to show that the property holds before the loop — since $[\![\bigwedge R \wedge \bigwedge T]\!]_{\mathscr{L}}$ never grows inside the loop body, preservation is easy. Thus, suppose that $c_1 \ [\![\bigwedge T]\!]_{\mathscr{L}} \ c_2$ where $T$ is initialized to $I$ as given above, and also that $\lfloor c_1 \rfloor \ \mathsf{reach}_\phi \ \lfloor c_2 \rfloor$. Assume towards a contradiction that $c_1 \in F$ if

and only if $c_2 \notin F$. In that case, $\lfloor c_1 \rfloor = t_{\mathsf{accept}}$ if and only if $\lfloor c_2 \rfloor \neq t_{\mathsf{accept}}$. Then, $\lfloor c_1 \rfloor^< \wedge \lfloor c_2 \rfloor^> \implies \bot \in T$, and thus $c_1 \, [\![ \bigwedge T ]\!]_{\mathscr{L}} \, c_2$ does *not* hold — a contradiction. Our assumption must have been wrong, and therefore $c_1 \in F$ iff $c_2 \in F$.

3. If $\rho$ is a symbolic bisimulation, then $\rho \vDash \bigwedge R \wedge \bigwedge T$. In this case, it again suffices to show that this property holds before the loop. To this end, let $c_1 \, [\![ \rho ]\!]_{\mathscr{L}} \, c_2$. For all $t_1 \, \mathsf{reach}_\phi \, t_2$ with $t_1 = t_{\mathsf{accept}} \iff t_2 \neq t_{\mathsf{accept}}$, we should argue that $c_1 \, [\![ t_1^< \wedge t_2^> ]\!]_{\mathscr{L}} \, c_2$ does *not* hold. Thus, suppose towards a contradiction that $c_1 \, [\![ t_1^< \wedge t_2^> ]\!]_{\mathscr{L}} \, c_2$. Now, if $c_1 \in F$, then $t_1 = t_{\mathsf{accept}}$; but then $t_2 \neq t_{\mathsf{accept}}$, meaning that $c_2 \notin F$. This contradicts that $\rho$ is a symbolic bisimulation with $c_1 \, [\![ \rho ]\!]_{\mathscr{L}} \, c_2$. Thus, $c_1 \, [\![ t_1^< \wedge t_2^> ]\!]_{\mathscr{L}} \, c_2$ does *not* hold — we are done.

When the loop terminates, all three invariants still hold. Specifically, (1) $[\![ \bigwedge R ]\!]_{\mathscr{L}}$ is preserved by $\delta$, (2) if $c_1 \, [\![ \bigwedge R ]\!]_{\mathscr{L}} \, c_2$ and $\lfloor c_1 \rfloor \, \mathsf{reach}_\phi \, \lfloor c_2 \rfloor$, then $c_1 \in F$ if and only if $c_2 \in F$, and (3) if $\rho$ is a symbolic bisimulation, then $\rho \vDash \bigwedge R$.

- Suppose the algorithm returns **true**. In that case, $\phi \vDash \bigwedge R$. We choose $\psi = \bigvee_{t_1 \, \mathsf{reach}_\phi \, t_2} (t_1^< \wedge t_2^>)$, and claim that $\psi \wedge \bigwedge R$ is a symbolic bisimulation. Clearly, both $[\![ \psi ]\!]_{\mathscr{L}}$ and $[\![ \bigwedge R ]\!]_{\mathscr{L}}$ preserve $\delta$ — the former by construction, the latter by the first loop invariant.

  To see that $\psi \wedge \bigwedge R$ is compatible with $F$, suppose $c_1 \, [\![ \psi \wedge \bigwedge R ]\!]_{\mathscr{L}} \, c_2$. In that case, we have $c_1 \, [\![ \psi ]\!]_{\mathscr{L}} \, c_2$, and so $\lfloor c_2 \rfloor \, \mathsf{reach}_\phi \, \lfloor c_2 \rfloor$. Also, since $c_1 \, [\![ \bigwedge R ]\!]_{\mathscr{L}} \, c_2$, we know that $c_1 \in F$ if and only if $c_2 \in F$ by the second loop invariant. Thus, $\psi \wedge \bigwedge R$ is indeed a symbolic bisimulation. Since $\phi \vDash \psi \wedge \bigwedge R$, we can conclude that $c_1 \, [\![ \phi ]\!]_{\mathscr{L}} \, c_2$ implies $L(c_1) = L(c_2)$ by Lemma 4.4.1.

- Suppose that $c_1 \, [\![ \phi ]\!]_{\mathscr{L}} \, c_2$ implies $L(c_1) = L(c_2)$. By Lemma 4.4.1 there exists some symbolic bisimulation $\psi$ such that $\phi \vDash \psi$. By the loop invariant, we know that $\psi \vDash \bigwedge R$. It then follows that $\phi \vDash \bigwedge R$, and thus the algorithm returns **true**. □

**Lemma 4.5.1.** *Let $\phi$ be a formula. The following are equivalent:*

1. *There exists a symbolic bisim. with leaps $\psi$ s.t. $\phi \vDash \psi$.*
2. *If $c_1 \, [\![\phi]\!]_{\mathscr{L}} \, c_2$, then $L(c_1) = L(c_2)$.*

*Proof.* The backward implication is straightforward. In this case, we note that $\phi \vDash \psi$ for some symbolic bisimulation $\psi$. Because any (symbolic) bisimulation is in particular a (symbolic) bisimulation with leaps, the claim follows.

For the other direction, suppose that $\psi$ is a symbolic bisimulation with leaps such that $\phi \vDash \psi$. We define $R$ as the smallest relation satisfying

$$\frac{c_1 \, [\![\psi]\!]_{\mathscr{L}} \, c_2 \qquad w \in \{0,1\}^*}{\delta^*(c_1, w) \; R \; \delta^*(c_2, w)}$$

Clearly, $R$ is closed under steps by construction.

We claim that $R$ is a bisimulation. It suffices to prove that for all $c_1 \, [\![\psi]\!]_{\mathscr{L}} \, c_2$ and $w \in \{0,1\}$, we have $\delta^*(c_1, w) \in F$ if and only if $\delta^*(c_2, w) \in F$. We proceed by induction on $|w|$.

In the base, $w = \epsilon$. We can then conclude that $\delta^*(c_1, w) = c_1 \in F$ if and only if $\delta^*(c_2, w) = c_2 \in F$, because $c_1 \, [\![\psi]\!]_{\mathscr{L}} \, c_2$ and $\psi$ is a symbolic bisimulation with leaps.

For the inductive step, let $|w| > 0$ and $n = \sharp(c_1, c_2)$, and assume that the claim holds for all $y \in \{0,1\}$ with $|y| < |w|$. On the one hand, if $|w| < n$, then necessarily $n > 1$, and thus $c_1, c_2 \notin F$. This means that $\delta^*(c_1, w), \delta^*(c_2, w) \notin F$, because the state component of those configurations does not change in the first $n$ steps. On the other hand, if $|w| \geq n$, then we write $w = xy$ with $|x| = n$. Now, since $[\![\psi]\!]_{\mathscr{L}}$ is a symbolic bisimulation with leaps we know that $\delta^*(c_1, x) \, [\![\psi]\!]_{\mathscr{L}} \, \delta^*(c_2, x)$. Finally, since $|y| < |w|$, we have

$$\delta^*(c_1, w) = \delta^*(\delta^*(c_1, x), y) \in F \iff$$
$$\delta^*(c_2, w) = \delta^*(\delta^*(c_2, x), y) \in F$$

by induction. □

## A.3 Code listings

```
parse_eth {
  extract(ether, 112);
  select(ether[0:0]) {
    0 ⇒ default_vlan
    1 ⇒ parse_vlan
  }
}
default_vlan {
  vlan := 0x0000;
  extract(ip, 160)
  goto parse_udp
}
parse_vlan {
  extract(vlan, 32);
  goto parse_ip
}
parse_ip {
  extract(ip, 160);
  goto parse_udp
}
parse_udp {
  extract(udp, 64);
  select(vlan[0:3]) {
    1111 ⇒ reject
    _    ⇒ accept
  }
}
```

Listing A.1: Ethernet stack parser with an optional VLAN tag.

```
parse_eth {
```

```
  extract(ether, 112);
  select(ether[96:111]) {
    0x86dd ⇒ parse_ipv6
    0x8600 ⇒ parse_ipv4
  }
}
parse_ipv6 {
  extract(ipv4, 288)
  goto accept
}
parse_ipv4 {
  extract(ipv6, 128);
  goto accept
}
```

Listing A.2: Sloppy parser for Ethernet and IP.

```
parse_eth {
  extract(ether, 112);
  select(ether[96:111]) {
    0x86dd ⇒ parse_ipv6
    0x8600 ⇒ parse_ipv4
    _   ⇒ reject
  }
}
parse_ipv6 {
  extract(ipv4, 288)
  goto accept
}
parse_ipv4 {
  extract(ipv6, 128);
  goto accept
```

```
}
```

Listing A.3: Strict parser for Ethernet and IP.

```
parse_0 {
  extract(T0, 8);
  extract(L0, 8);
  select(T0, L0) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (_, 0x01) ⇒ goto parse_v01
    (_, 0x02) ⇒ goto parse_v02
    (_, 0x03) ⇒ goto parse_v03
    (_, 0x04) ⇒ goto parse_v04
    (_, 0x05) ⇒ goto parse_v05
    (_, 0x06) ⇒ goto parse_v06
  }
}
parse_v01 {
  extract(scratch, 8);
  v0 ← scratch ++ v0[7:47]
  goto parse_1
}
parse_v02 {
  extract(scratch, 16);
  v0 ← scratch ++ v0[15:47]
  goto parse_1
}
parse_v03 {
  extract(scratch, 24);
  v0 ← scratch ++ v0[23:47]
  goto parse_1
}
```

```
parse_v04 {
  extract(scratch, 32);
  v0 ← scratch ++ v0[31:47]
  goto parse_1
}
parse_v05 {
  extract(scratch, 40);
  v0 ← scratch ++ v0[39:47]
  goto parse_1
}
parse_v06 {
  extract(v0, 48);
  goto parse_1
}
parse_1 {
  extract(T1, 8);
  extract(L1, 8);
  select(T1, L1) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (_, 0x01) ⇒ goto parse_v11
    (_, 0x02) ⇒ goto parse_v12
    (_, 0x03) ⇒ goto parse_v13
    (_, 0x04) ⇒ goto parse_v14
    (_, 0x05) ⇒ goto parse_v15
    (_, 0x06) ⇒ goto parse_v16
  }
}
parse_v11 {
  extract(scratch, 8);
  v1 ← scratch ++ v1[7:47]
  goto parse_2
```

```
}
parse_v12 {
  extract(scratch, 16);
  v1 ← scratch ++ v1[15:47]
  goto parse_2
}
parse_v13 {
  extract(scratch, 24);
  v1 ← scratch ++ v1[23:47]
  goto parse_2
}
parse_v14 {
  extract(scratch, 32);
  v1 ← scratch ++ v1[31:47]
  goto parse_2
}
parse_v15 {
  extract(scratch, 40);
  v1 ← scratch ++ v1[39:47]
  goto parse_2
}
parse_v16 {
  extract(v1, 48);
  goto parse_2
}
parse_2 {
  extract(T2, 8);
  extract(L2, 8);
  select(T2, L2) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (_, 0x01) ⇒ goto parse_v21
```

```
    (_, 0x02) ⇒ goto parse_v22
    (_, 0x03) ⇒ goto parse_v23
    (_, 0x04) ⇒ goto parse_v24
    (_, 0x05) ⇒ goto parse_v25
    (_, 0x06) ⇒ goto parse_v26
  }
}
parse_v21 {
  extract(scratch, 8);
  v2 ← scratch ++ v2[7:47]
  goto accept
}
parse_v22 {
  extract(scratch, 16);
  v2 ← scratch ++ v2[15:47]
  goto accept
}
parse_v23 {
  extract(scratch, 24);
  v2 ← scratch ++ v2[23:47]
  goto accept
}
parse_v24 {
  extract(scratch, 32);
  v2 ← scratch ++ v2[31:47]
  goto accept
}
parse_v25 {
  extract(scratch, 40);
  v2 ← scratch ++ v2[39:47]
  goto accept
}
```

```
parse_v26 {
  extract(v2, 48);
  goto accept
}
```

Listing A.4: Generic IP options parser.

```
parse_0 {
  extract(T0, 8);
  extract(L0, 8);
  select(T0, L0) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (0x44, 0x06) ⇒ goto parse_stamp0
    (_, 0x01) ⇒ goto parse_v01
    (_, 0x02) ⇒ goto parse_v02
    (_, 0x03) ⇒ goto parse_v03
    (_, 0x04) ⇒ goto parse_v04
    (_, 0x05) ⇒ goto parse_v05
    (_, 0x06) ⇒ goto parse_v06
  }
}
parse_stamp0 {
  extract(ptr0, 8);
  extract(over0, 4);
  extract(flag0, 4);
  extract(time0, 32);
  goto parse_1
}
parse_v01 {
  extract(scratch, 8);
  v0 ← scratch ++ v0[7:47]
  goto parse_1
```

```
}
parse_v02 {
  extract(scratch, 16);
  v0 ← scratch ++ v0[15:47]
  goto parse_1
}
parse_v03 {
  extract(scratch, 24);
  v0 ← scratch ++ v0[23:47]
  goto parse_1
}
parse_v04 {
  extract(scratch, 32);
  v0 ← scratch ++ v0[31:47]
  goto parse_1
}
parse_v05 {
  extract(scratch, 40);
  v0 ← scratch ++ v0[39:47]
  goto parse_1
}
parse_v06 {
  extract(v0, 48);
  goto parse_1
}
parse_1 {
  extract(T1, 8);
  extract(L1, 8);
  select(T1, L1) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (0x44, 0x06) ⇒ goto parse_stamp1
```

```
    (_, 0x01) ⇒ goto parse_v11
    (_, 0x02) ⇒ goto parse_v12
    (_, 0x03) ⇒ goto parse_v13
    (_, 0x04) ⇒ goto parse_v14
    (_, 0x05) ⇒ goto parse_v15
    (_, 0x06) ⇒ goto parse_v16
  }
}
parse_stamp1 {
  extract(ptr1, 8);
  extract(over1, 4);
  extract(flag1, 4);
  extract(time1, 32);
  goto parse_2
}
parse_v11 {
  extract(scratch, 8);
  v1 ← scratch ++ v1[7:47]
  goto parse_2
}
parse_v12 {
  extract(scratch, 16);
  v1 ← scratch ++ v1[15:47]
  goto parse_2
}
parse_v13 {
  extract(scratch, 24);
  v1 ← scratch ++ v1[23:47]
  goto parse_2
}
parse_v14 {
  extract(scratch, 32);
```

```
  v1 ← scratch ++ v1[31:47]
  goto parse_2
}
parse_v15 {
  extract(scratch, 40);
  v1 ← scratch ++ v1[39:47]
  goto parse_2
}
parse_v16 {
  extract(v1, 48);
  goto parse_2
}
parse_2 {
  extract(T2, 8);
  extract(L2, 8);
  select(T2, L2) {
    (0x00, 0x00) ⇒ goto accept
    (0x01, 0x00) ⇒ goto accept
    (0x44, 0x06) ⇒ goto parse_stamp2
    (_, 0x01) ⇒ goto parse_v21
    (_, 0x02) ⇒ goto parse_v22
    (_, 0x03) ⇒ goto parse_v23
    (_, 0x04) ⇒ goto parse_v24
    (_, 0x05) ⇒ goto parse_v25
    (_, 0x06) ⇒ goto parse_v26
  }
}
parse_stamp2 {
  extract(ptr2, 8);
  extract(over2, 4);
  extract(flag2, 4);
  extract(time2, 32);
```

```
  goto accept
}
parse_v21 {
  extract(scratch, 8);
  v2 ← scratch ++ v2[7:47]
  goto accept
}
parse_v22 {
  extract(scratch, 16);
  v2 ← scratch ++ v2[15:47]
  goto accept
}
parse_v23 {
  extract(scratch, 24);
  v2 ← scratch ++ v2[23:47]
  goto accept
}
parse_v24 {
  extract(scratch, 32);
  v2 ← scratch ++ v2[31:47]
  goto accept
}
parse_v25 {
  extract(scratch, 40);
  v2 ← scratch ++ v2[39:47]
  goto accept
}
parse_v26 {
  extract(v2, 48);
  goto accept
```

```
}
```

Listing A.5: Generic IP options parser.

## BIBLIOGRAPHY


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM POPL*, pages 113–126, 2014.

[4] Andrew W. Appel. Verified software toolchain. In *Proc. of European Symposium on Programming*, ESOP, pages 1–17, 2011.

[5] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, 2017.

[6] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[7] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Proc. of the International Conference on Certified Programs and Proofs*, CPP, pages 135–150, 2011.

[8] Internet Assigned Numbers Authority. Internet protocol version 4 (ipv4) parameters, 2018.

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of Computer Aided Verification*, CAV, pages 171–177, 2011.

[10] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. of the 8th International Workshop on Satisfiability Modulo Theories*, volume 13 of *SMT*, pages 14–14, 2010.

[11] Ryan Becket, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *SIGCOMM*, pages 155–168, 2017.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

[13] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the Sockets API. *JACM*, 66(1):1:1–1:77, December 2018.

[14] Nikolaj Bjorner and Karthick Jayaraman. Checking cloud contracts in Microsoft Azure. In *ICDCIT*, pages 21–32. Springer-Verlag, 2015.

[15] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, page 457–468, 2013.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

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

[18] Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. Minimal model generation. In *Proc. of the 2nd International Workshop on Computer Aided Verification*, CAV, pages 197–203, 1991.

[19] Ahmed Bouajjani, Jean-Claude Fernandez, Nicolas Halbwachs, and Pascal Ray-

mond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, 1992.

[20] Amar Bouali and Robert de Simone. Symbolic bisimulation minimisation. In *Proc. of the 4th International Workshop on Computer Aided Verification*, CAV, pages 96–108, 1992.

[21] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lucius J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[22] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 1027–1040, 2019.

[23] Cisco Systems. Cisco DNA analytics and assurance, 2018. Available at `https://www.cisco.com/c/en/us/solutions/enterprise-networks/dna-analytics-assurance.html`.

[24] The P4 Language Consortium. *P4 Language Specification, Version 1.2.2*, May 2021. Available at `https://p4.org/p4-spec/docs/P4-16-v1.2.2.html`.

[25] The Coq Development Team. *The Coq Reference Manual, version 8.14*, October 2021. Available electronically at `http://coq.inria.fr/doc`.

[26] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. of Automatic Verification Methods for Finite State Systems*, pages 365–373, 1989.

[27] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1–4):423–453, jun 2018.

[28] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Available at `http://hdl.handle.net/1842/13555`.

[29] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. In *Proc. of the 33rd International Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, pages 79–99, 2018.

[30] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of*

*the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 337–340, 2008.

[31] Christian Dehnert, Joost-Pieter Katoen, and David Parker. Smt-based bisimulation minimisation of markov models. In *Proc. of the 14th International Workshop on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 28–47, 2013.

[32] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages*, 3(ICFP), July 2019.

[33] Salem Derisavi. A symbolic algorithm for optimal markov chain lumping. In *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 139–154, 2007.

[34] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[35] Catherine Dodge and Stephen Quigg. A simpler way to assess the network exposure of ec2 instances: Aws releases new network reachability assessments in amazon inspector, November 2018. Archived at `https://web.archive.org/web/https://aws.amazon.com/blogs/security/amazon-inspector-assess-network-exposure-ec2-instances-aws-network-reachabi`

[36] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for p4 data planes, 2020.

[37] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.

[38] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 950–965, 2022.

[39] Yuan Feng, Yuxin Deng, and Mingsheng Ying. Symbolic bisimulation for quantum processes. *ACM Transactions on Computational Logic*, 15(2), May 2014.

[40] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, pages 469–483, 2015.

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

[42] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT compilers for in-kernel DSLs. In *CAV*, 2020.

[43] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, pages 300–313, 2016.

[44] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Proc. of the 9th ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, pages 13–24, 2013.

[45] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Proc. of the International Workshop on Types for Proofs and Programs*, TYPES, page 39–59, 1994.

[46] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the posix shell. In *POPL*, 2020.

[47] Armaël Guéneau, Magnus O Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for cakeml. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*, pages 584–610. Springer, 2017.

[48] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, pages 483–494, 2013.

[49] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, 2010.

[50] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[51] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *PLDI*, pages 185–200, 2017.

[52] Matthew Hennessy and Huimin Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[53] Stefan Heule, Konstantin Weitz, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. Leveraging p4 to automatically validate networking switches, September 2019. Presentation at ONF Connect. Slides available at `https://www.opennetworking.org/wp-content/uploads/2019/09/2.30pm-Stefan-Heule-P4-Presentation.pdf`.

[54] Mukesh Hira and LJ Wobker. Improving network monitoring and management with programmable data planes. P4 Language Consortium Blog, September 2015. Available at `https://p4.org/p4/inband-network-telemetry/`.

[55] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Proceedings of an International Symposium on the Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[56] John Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71-114, Cornell University, Ithaca, NY, December 1971.

[57] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proc. of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 193–206, 2013.

[58] IEEE Computer Society. Ieee standard for local and metropolitan area network–bridges and bridged networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pages 1–1993, 2018.

[59] The MathWorks Inc. Matlab, 2022.

[60] Information Technology – Programming Languages – C. Standard, International Organization for Standardization, Geneva, CH, 2018.

[61] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *NSDI*, pages 35–49, 2018.

[62] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *SOSP*, pages 121–136, 2017.

[63] Jacques-Henri Jourdan and François Pottier. A simple, possibly correct lr parser for c11. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(4):1–36, 2017.

[64] Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, page 22–39. Springer-Verlag, 1987.

[65] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, PODC, page 228–240, 1983.

[66] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified credible compilation for llvm. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 631–645, 2018.

[67] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.

[68] Ali Kheradmand and Grigore Rosu. P4k: A formal semantics of p4 and applications. 2018.

[69] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[70] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, July 2009.

[71] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[72] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully emulating large production networks. In *SOSP*, pages 599–613, 2017.

[73] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: Practical

verification for programmable data planes. In *ACM SIGCOMM*, pages 490–503, 2018.

[74] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan 2021.

[75] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, pages 290–301, 2011.

[76] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proc. of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS, page 122–126, 1987.

[77] Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjørner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in p4 networks. Technical Report MSR-TR-2016-65, September 2016.

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

[79] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129–154. Princeton University Press, 2016.

[80] Malcolm Mumme and Gianfranco Ciardo. A fully symbolic bisimulation algorithm. In *Proc. of the 5th International Workshop on Reachability Problems*, RP, pages 218–230, 2011.

[81] George C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI, page 83–94, 2000.

[82] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. Verification of p4 programs in feasible time using assertions. In *Proc. of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT, page 73–85, 2018.

[83] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[84] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas.

p4pktgen: automated test case generation for P4 programs. In *ACM SOSR*, pages 5:1–5:7, 2018.

[85] P4 Tutorial. `https://github.com/p4lang/tutorials`, 2023.

[86] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[87] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356, June 2015.

[88] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, January 2000.

[89] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[90] Jon Postel. User Datagram Protocol. RFC 768, August 1980.

[91] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[92] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure Zero-Copy parsers for authenticated message formats. In *28th USENIX Security Symposium*, USENIX Security, pages 1465–1482, August 2019.

[93] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *OSDI*, November 2020.

[94] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

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

[96] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[97] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, January 2010.

[98] Christian Skalka, John Ring, David Darias, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. Proof carrying network code. In *ACM CCS*, pages 1115–1129, November 2019.

[99] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, ITP, pages 419–434, 2010.

[100] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP), Jul 2019.

[101] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *SIGCOMM*, 2018.

[102] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[103] Aldo Svaldi. A single network card caused centurylink's nationwide outage. The Denver Post, January 2019. Archived at `https://web.archive.org/web/20190202225936/https://www.denverpost.com/2019/01/11/centurylink-network-outage-denver/`.

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

[105] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[106] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: A practically usable verification system for production-scale programmable data planes. In *Proc. of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM, page 17–32, 2021.

[107] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, page 316–326, New York, NY, USA, 2009.

[108] Arun Viswanathan, Eric C. Rosen, and Ross Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[109] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–7, 2012.

[110] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *OSDI*, pages 33–47, 2014.

[111] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages*, 5(ICFP), August 2021.

[112] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. Spacesearch: A library for building and verifying solver-aided tools. *Proceedings of the ACM on Programming Languages*, 1(ICFP), August 2017.

[113] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proc. of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP, page 71–84, 2020.