VERIFIED CONFIGURATION OF PROGRAMMABLE NETWORKS

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

Eric Hayden Campbell

December 2024

VERIFIED CONFIGURATION OF PROGRAMMABLE NETWORKS

Eric Hayden Campbell, Ph.D.

Cornell University 2024

Reasoning about network programs is challenging because of how they divide labor: the control plane computes high level routes through the network and compiles them to device configurations, while the data plane uses these configurations to realize the desired forwarding behavior. In practice, the correctness of the data plane often assumes that the configurations generated by the control plane will satisfy complex specifications. These specifications are either missing or maintained in complex English language documents, which makes correctly configuring devices hard.

This thesis tackles this problem from three angles. First, we present algorithms for computing control plane interface specifications that ensure the safety of the data plane. These specifications can be used to improve the safety and quality of both the control plane and of the data plane. Then, we show how to automatically generate configurations for data plane programs, and finally, we conclude with a semantic framework for programming and relational verification of pairs of configurable programs.

BIOGRAPHICAL SKETCH

Eric Hayden Campbell was born at the University of Chicago hospital in 1995. Raised primarily in San Jose, California, he and his family moved to Amsterdam in 2011. He graduated from the International School of Amsterdam in 2013, at which point, he moved back to California to attend Pomona College. There he fell in love with computer science, discrete mathematics, and formal logic, receiving his bachelor's degrees in Mathematics and in Computer Science in 2017.

For Priya, who never pulled punches

ACKNOWLEDGEMENTS

First and foremost, thanks to my advisor, Nate. Over the years, Nate has put up with my crackpot ideas and guided me through numerous eleventh-hour paper pushes. I thank him for his unwavering support and guidance.

I would also be remiss if I didn't thank Hossein Hojjat for being my formal methods guru: introducing me to synthesis and the formal methods community at large.

Thanks to Mark Moeller for moral support, running support, and for never being too good for a free banana. Thanks to Ryan Doenges for encouraging me, by example, to only do exactly what I wanted to do. Thanks to Jonathan DiLorenzo for mentoring me in health, wine, and cheese.

Thanks to Ryland Bednarek, Kiran Tomlinson, Mario Sanchez, Nora Murphy, Morgan Shelton, Peter Chwazik, and Marty Heresniak, for pushing my musical development, and giving me a creative restful outlet that has, through the years, supported my main academic endeavors.

And to Jane Lu, thanks for the endless dancing, singing, and outdoor adventures.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vi
List of Tables ix
List of Figures x

I The Problem of Interfaces **1**

1 Managing Changing Programs **2**
1.1 A Subprime Meridian 3
1.2 Improving P4 as a Specification Language 4
1.3 Leveraging Specifications 6
1.4 Summary of Contributions 8
1.5 Attribution and Acknowledgements 10

2 Data Plane Programming **12**
2.1 Data Plane Programs Have Bugs 18
2.1.1 Parser Bugs 18
2.1.2 Control Bugs 20
2.1.3 Table Reads Bugs 21
2.1.4 Table Action Bugs 23
2.1.5 Default Action Bugs 25

3 Static Data Plane Analysis and Control Assumptions **26**
3.1 A “Featherweight” P4 29
3.1.1 Syntax 32
3.1.2 Type System 35
3.1.3 Operational Semantics 40
3.1.4 Safety of SafeP4 43
3.2 Experience (Evaluation) 45
3.2.1 Overview of Bugs in the Wild 46
3.2.2 P4Check in Action 48
3.2.3 Overhead 53

4 Data Plane Verification and the Control Plane Interface **55**
4.1 The Guarded Command Language 56
4.2 Symbolic Compilation 59
4.3 Modeling P4 in GCL 60
4.3.1 Headers 60
4.3.2 Parsing 61
4.3.3 Hash Functions 62

4.3.4 Stateful Operations 62
4.3.5 Match-Action Tables 62
4.4 A First Attempt: Manual Control Interface Specs 63

II Control Interface Specifications 66

5 Computing Precise Control Interface Specifications 67
5.1 Background and Motivation 71
5.1.1 Inference of Control Interface Specifications 75
5.1.2 Previous Work 76
5.2 Modeling 77
5.2.1 Theories of Fixed-width Bitvectors 78
5.2.2 Syntax and Semantics of the Guarded Pipeline Language (GPL) 79
5.2.3 Modeling Tables as Uninterpreted Functions 82
5.2.4 Symbolic Compilation 84
5.3 Computing Efficiently Control-Monitorable Sentences 89
5.3.1 QE Computes Precise ci-specs 91
5.3.2 Precise ci-spec Inference in ECMS Solves QE 95
5.4 Programmatic QE 96
5.4.1 Paths Produce Smaller QE Problems 97
5.4.2 A Path-Based Iterative Strengthening Algorithm 99
5.5 Specifications for Data Planes 101
5.6 Implementation 103
5.7 Experience 104
5.7.1 Capisce in Practice 105
5.7.2 True Data-Plane Bugs 108
5.7.3 Bugs Found by Inspecting ci-specs 110
5.7.4 Analyzing Path Decomposition 112
5.7.5 Limitations 114
5.7.6 Comparison to `bf4` 116

III Verified Configuration 119

6 Automatically Configuring the Data Plane 120
6.1 Background and Motivation 125
6.2 Control Plane Synthesis 131
6.2.1 Basic Definitions and Verification 131
6.2.2 Synthesizing Candidates via Sketches 135
6.2.3 Counterexample-Guided Search 136
6.2.4 Synthesis Algorithm 137
6.2.5 Formal Properties 139

6.3 A Scalable Solution: Incremental Synthesis . . . 140
6.3.1 Single Counterexample-Guided Search . . . 140
6.3.2 Incremental Synthesis Algorithm . . . 143
6.3.3 Formal Properties . . . 144
6.4 Heuristics and Optimizations . . . 145
6.4.1 Exploiting Incrementality . . . 145
6.4.2 Model-Finding Heuristics . . . 147
6.5 Implementation . . . 148
6.6 Evaluation . . . 149
6.6.1 Retargeting Study . . . 151
6.6.2 Network Emulation . . . 153
6.6.3 Case Study: Trellis & ONOS . . . 153
6.6.4 Microbenchmarks . . . 155
6.7 Limitations and Future Work . . . 156
6.8 Conclusion . . . 157

7 Relational Hoare Lenses 159
7.1 Overview . . . 162
7.2 Basic Definitions for Relational Hoare Lenses . . . 165
7.2.1 Specifications for lenses . . . 168
7.2.2 Relational Hoare Lenses . . . 170
7.3 Relational Program Logics . . . 173
7.3.1 Relational Hoare Logic . . . 175
7.3.2 Reasoning about Programs with RHLenses . . . 179
7.4 Implementation . . . 183
7.5 Case Study: Network Data Plane Programs . . . 189
7.5.1 Source Program . . . 190
7.5.2 Action Decompose . . . 192
7.5.3 Metadata Decomposition . . . 194
7.5.4 Early Validation . . . 195

8 Related Work 197

9 Conclusion 202
9.1 Future Work . . . 204

Bibliography 206

LIST OF TABLES

5.1 Experience with using Capisce to check Header Validity on a broad range of P4 programs. 106
5.2 Experience with using Capisce to check Determined Forwarding on a broad range of P4 programs. 107

LIST OF FIGURES

2.1 Abstract forwarding model. 13
2.2 (Left) Header formats and parse graph that extracts an Ethernet header optionally followed by VLAN and/or IPv4 headers. (Right) P4 code implementing the same parser. 14
2.3 P4 tables. `forward` reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance, and calls one of its actions: `nop`, `next_hop`, or `remove`. 15
2.4 P4 actions. 17
2.5 Left: unsafe code in NetHCF; Right: our type-safe fix; Bottom: common code. 19
2.6 Left: unsafe code in NetCache; Right: our type-safe fix; Bottom: Common code 20
2.7 Left: a table in `switch.p4` with unprotected conditional reads; Right: our type-safe fix. 22
2.8 Left: unsafe code in `switch.p4`; Right: our type-safe fix. 24
2.9 Left: unsafe code in NetCache; Right: our type-safe fix. 25

3.1 Syntax of SafeP4 33
3.2 Semantics of header types (left) and auxiliary functions (right). . . 36
3.3 Command typing rules for SafeP4 37
3.4 Action typing rule for SafeP4 39
3.5 Expression typing rules for SafeP4 39
3.6 Selected rules of the operational semantics of SafeP4; the elided rules are standard. 41
3.7 Selected rules of the operational semantics for expressions. 42
3.8 The *Entailment* relation between header instances and header instance types 43
3.9 Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported at the top of each stacked bar. 46
3.10 Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar 46
3.11 Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet` 49
3.12 Warnings printed after fixing `switch.p4`'s reads bug (top), and its actions bug (bottom) 50

4.1 Syntax (left) and semantics (right) of the Guarded Command Language GCL. 56
4.2 Bitvector Arithmetic: expressions (left) and logical formulae (right) 58

5.1 The control plane generates configs that define data plane behavior. 67

5.2 An example data plane pipeline program (right) and with an asserted ci-spec (bottom right). Capisce computes a precise ci-spec (center), which ensures that the pipeline satisfies the spec. If the control plane (left) installs a bad config (top), it is rejected. Safe configs, like the one shown on the bottom, are accepted and can be safely installed into the pipeline program. 72
5.3 Bitvector Theories. The syntax of UFBV formulae (left) and expressions (middle). The classification of bitvector theories (above right), depending on whether they allow quantifiers ($\forall$) and/or uninterpreted functions (F). The semantics of bitvector expressions are standard. 78
5.4 Syntax (left) and semantics (right) of Guarded Pipeline Language GPL($\mathcal{T}$) over a bitvector theory $\mathcal{T}$. Highlighted variants only occur in GPL($\mathcal{T}$); the other variants are Guarded Command Language GCL($\mathcal{T}$). 80
5.5 Path coverage over time for Header Validity analysis of programs with fewer than 100k paths. 112
5.6 Comparing analysis capabilities of `bf4` and Capisce w.r.t time (bottom) and bugs controlled (top). Note the logarithmic y-axes on the time charts. 117

6.1 Avenir maps control plane operations for an abstract pipeline into corresponding operations for a target using sketch-based synthesis. The synthesis loop alternates between verifying the correctness of a candidate implementation and learning from counterexamples to generate a better one; the holes (e.g., $?_5$) in the target sketch denote missing values that are filled in using an SMT solver. 124
6.2 Pipelines used in example scenario. 125
6.3 **The Status Quo** Manual translations from Pipeline 1 to Pipelines 2 and 3. Avenir automates theses translations entirely 125
6.4 **The Status Quo**: Manual translations in pseudocode from "one big table" (OBT) to Pipelines 1 through 3. 126
6.5 Dynamic Configurations used in example scenario. $Pipe_2$ is annotated with "holes" to be filled in. During synthesis, Avenir solves for these unknowns and concludes that $?_1$ = ABB28FC, $?_2$ = set_meta(5), $?_3 = 5$, $?_4$ = set_out(5). 129
6.6 Pipeline syntax. Actions vary under starred variants 132
6.7 Summary of holes used in sketching. 135
6.8 The model function. In the above, the vector $\overline{x}$ is all of the non-hole variables that occur in the formula. 137
6.9 Simple Algorithm for Control Plane Synthesis. 137
6.10 Basic Sketch for $Pipe_1$: Satisfiable for packets that hit L2's first row and L3's second. 138

6.11 Incremental Sketch for $Pipe_1$: Unsatisfiable for packets that hit L2's first row and L3's second, which triggers backtracking, remembering that the previously-synthesized edit was incorrect. 138
6.12 The model' function computes edits to physical state (p, σ) to accomodate the counterexample χ. The oracle soundly restricts the search space. 142
6.13 The incremental backtracking CEGIS algorithm. 142
6.14 Retargeting case study: solid lines show cold-start completion %; dotted lines show hot-start completion %. 150
6.15 Proportion of all pairs of 64 hosts connected in a star topology that have completed a successful IPv4 ping. 150
6.16 Completion graph for mapping 40k fabric.p4 IPv6 route insertions onto bcm.p4; ONOS takes around 15 min. 151
6.17 Program bits vs time to translate 100 edits. The vertical lines estimate the sizes of common router programs. 155
6.18 Classifier Scaling. We fixed the number of 32-bit output variables to 8, and varied the number of keys. 155

7.1 A lens synchronizing abstract and concrete views of a packet forwarding pipeline. The relation Θ describes the relation maintained by the lens's $putR$ and $putL$ functions. 163
7.2 A judgment defining whether a configuration is valid *w.r.t.* a schema 173
7.3 Syntax (left), denotational semantics (right) , and auxiliary sets (bottom) for WHILE 176
7.4 The syntax (left) and semantics of relational expressions (middle) and formulae (right) 177
7.5 Structural axioms and inference rules for closed programs [15, 9] . . 179
7.6 Correpondence between the RHL proof system and RHLens combinators 183
7.7 Summary of the core data structures and smart constructors in Spectacle. 184
7.8 Modules describing Lenses (right) and the configurations they act upon (left) 185

Part I

The Problem of Interfaces

CHAPTER 1

MANAGING CHANGING PROGRAMS

The Internet was built on standards. Standardization bodies produce specification documents that describe numerous standardized protocols for enabling communication between devices. These protocols specify intricate message formats, semantics, and patterns. Having a robust set of protocols with agreed-upon semantics enables decentralized growth of computer networks. That is, an engineer who faithfully implements a standardized communication protocol, such as BGP, can generally believe that their implementation will be able to communicate with any other BGP-enabled device.

Modern networking architectures promote a different set of values. From the reliable model of rigid standardization, design patterns like software-defined networking (SDN) and deep network programability [52] permit flexibility and agility in network design. For instance, the programming language P4 allows engineers to ***p****rogram* ***p****rotocol-independent* ***p****acket* ***p****rocessors* [20]. That is, engineers are able to define their own packet-level communication formats and forwarding patterns to suit their specific needs.

However, this preponderance of data formats is dangerous for system reliability. As any standardization committee will tell you, the path to "rough consensus and working code" is a long one, requiring many iterations before the protocol is standardized. Indeed, the modern Internet protocol suite is the result of decades of debate and scores of revisions. Similarly, bespoke packet processing logic typically requires many iterations before it reaches a stable state. As we'll see, each of these changes to the packet processing logic can have significant ramifications throughout the networking software stack.

Rather than yearn for the good old days of standardization, this dissertation proposes formal tools that help us move *with* the constant ebb and flow of software development for packet processors. If we can formally understand the ramifications of various changes (Chapter 5), abstract changes away entirely (Chapter 6), and/or reason about changes formally (Chapter 7), we can more robustly manage the variation and evolution of packet processing software.

1.1 A Subprime Meridian

The computer network is divided by an architectural meridian: the interface between the control plane and the data plane. The control plane does the complex computational work: computing routes through the network to ensure requests get served, deciding which potentially malicious agents to block to ensure they cannot access sensitive data, or balancing the network load between hosts to ensure that no one gets overwhelmed. Meanwhile, the data plane—which is a collection of hardware that is highly-specialized to rapidly and efficiently transform and transfer internet data packets—realizes the control plane's policies. We use the term *switch* to refer to an abstract data plane forwarding device. To enact high-level policies, the control plane configures the data plane's behavior using the *control plane interface.*

Unfortunately, the control plane interface is poorly specified. Traditionally, hardware vendors release switches alongside lengthy English language documents that describe the control plane interface. However, these specifications are often insufficient, leaving significant behaviors undefined, underspecified, or simply incorrect.

Network engineers have been using P4 to shore up the robustness of the control-plane interface. Optimistically, network engineers write P4 code that is compiled directly to a configurable hardware format. The P4 code specifies both structure of the interface, and the forwarding logic that relies on that interface. Conceptually, the P4 code implements the *semantics* of the interface. However, after Intel's recent cancelation of its flagship P4-enabled switch line, Tofino, using P4 to implement real computer networks is less common.

A more widely applicable use case of P4, is as a *specification* language. Network engineers, especially at Google, have been using P4 to specify fixed-function, i.e. non-programmable, switches from vendors like Broadcomm [2]. These specification programs can be used bi-directionally, both to characterize correct behavior for the data plane switch it models and to characterize correct behavior for the control plane. This has enabled engineers to perform automatic verification [72, 108] and to generate concrete packets and configurations to use as test cases for the data plane [97].

1.2 Improving P4 as a Specification Language

Unfortunately, P4 was designed for *programming*, not *specification*, which has caused some growing pains. A very sensible language design decision—especially for a language that is meant to be amenable to many different hardware targets—is to allow compiler designers significant freedom in interpreting the language. In the specification, this is characterized as *undefined behavior*. Largely, the P4 language limits undefined behavior, but there are a few places it can arise.

Undefined behavior presents a challenge to using P4 as a specification language, because its presence is generally not apparent to the programmer. In Section 2.1

we discuss a collection of bugs in real P4 programs where possibly-uninitialized data was accessed—in P4, reading uninitialized data returns an undefined value.

In Chapter 3 we define a type system for an "featherweight" version of P4 called SafeP4 that avoids reading unintialized data. We use an occurence-style type system, and prove that our analysis is sound and complete. We also analyze a collection of P4 programs and propose a taxonomy of of code repairs that corresponds to the taxonomy of bugs from Section 2.1.

We can generalize these obervations beyond just avoiding undefined behavior. In general, a P4 program may have its own *correctness specification* that must be satisfied: e.g. *IPv4* and *IPv6* can never exist simultaneously in a single packet, or an emitted packet's time-to-live (TTL) field is never zero (which should indicate that the packet should be dropped, and an error packet returned to the sender).

Ideally, P4 programs-as-specifications would be written to satisfy the data plane correctness specification for every possible way that the control plane could configure it. However, in practice, programmers often make assumptions about the content of the configurations (i.e. that they must adhere to complex invariants). As a result, verification tools require assumptions about the behavior of the control plane. For instance, SafeP4 (Chapter 3) makes a collection of assumptions that the control plane is locally-sensible. That is, we assume that the control plane never reads invalid data. However, if even in the presence of these assumptions, the controller can configure the tables in a way that allows invalid data to be read, the program will be rejected by the type system.

Taking a step back, we want a general framework for characterizing limitations on the control plane. Existing work on `p4-constraints`, and `p4v` (see Chapter 4) has proposed using of first-order formulae to constrain the control plane interface. These *control plane interface specs* (ci-specs) are very useful; they can be used to: runtime-monitor the control plane, fuzz-test the control plane and the data plane, or even use them in a formal verification context. However, network engineers are not used to writing specifications.

In Chapter 5 we propose Capisce, a framework for inferring ci-specs automatically—taking as input a switch program and its correctness specification. The ci-specs ψ computed by Capisce are *precise*: a configuration satisfies ψ if and only if there are no reachable bugs in the switch program. They are also *efficiently control-monitorable*, which means that for a fixed configuration, evaluating ψ is polynomial.

Having precise ci-specs closes the loop on using P4 as a programming-and-specification language for switches. Now programmers have an executable P4 program that gives precise semantics to the interface, and a set of constraints—the ci-spec—that describes the correct use of the program.

1.3 Leveraging Specifications

We'll see that once we can precisely characterize the control plane interface, we can manage the variability and diversity that is inherent to computer networks. Most network operators build networks to support a varied collection of hardware, which results in devices with heterogeneous feature sets. This heterogeneity manifests as complexity throughout the control plane.

To manage device heterogeneity, network engineers can leverage abstraction—

designing a unifying logical interface to support the motley collection of physical switches. Of course, both the unifying logical interface and the corresponding switches and can be specified as P4 programs. For example, in the Open Network Operating System (ONOS [17]), the high-level interface is specified using a P4 program called `fabric.p4`, which is then mapped down to an array of target devices, which are either implemented or specified in P4.

The handwritten mapping code that synchronizes the configurations of logical and target programs is expensive to develop and error prone to build and maintain. To make matters worse, and every incremental change to a switch program or minor difference between switches can result in wide-sweeping changes. ONOS engineers shared an example (more in Chapter 6) of a seemingly trivial data plane change that broke abstraction boundaries and propagated all the way to the network application logic.

In Part III we leverage code as specifications to synchronize the interface of the logical and target programs in a verified manner. Concretely, we want to ensure with synchronized configs, the logical and target programs are equivalent.

In Chapter 6 we present Avenir, an automated synthesis tool that translates configs in an equivalence-preserving way. Importantly, our algorithm is incremental, which allows it to be fast and avoid significant recomputation work when the configurations get sizable. Alongside formal proofs of correctness, our evaluation on handcrafted benchmarks, industrial programs, and microbenchmarks shows that Avenir is scalable and expressive.

However, as with any synthesis tool, there will be times when Avenir cannot find a solution in a reasonable amount of time—even when one exists. So some degree of manual intervention is desirable. Further, Avenir's synchronization only works in one direction, and there are on-switch gadgets that can modify the configurations unbeknownst to the control plane.

The final chapter, Chapter 7 provides a semantic framework for reasoning about synchronizing pairs of reconfigurable programs. Synthesizing decades of work in bidirectional programming [50] and relational hoare logic [16], we propose *relational hoare lenses* (RHLenses), which allow us to reason about relational program properties, like equivalence, in the presence of synchronization code—the *lenses*. Indeed, relational hoare lenses has more general applications than just synchronizing network dataplanes, including in security, databases, and operating systems, but for this thesis we focus on this networking application, showing that we can use RHLenses to model the handcrafted benchmarks we used to evaluate Avenir.

1.4 Summary of Contributions

We summarize the contributions of this thesis as follows:

- Chapter 3 describes SafeP4, a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy essential safety guarantees. SafeP4 is equipped with a formal semantics and a static type system that statically guarantees header validity—a common source of safety bugs, according to our analysis of real-world P4 programs. Statically ensuring header validity is challenging because the set of valid headers can be modified at runtime, making it a dynamic program property. Our type system achieves static safety by using a form of path-sensitive reasoning that tracks dynamic information from conditional statements, routing tables, and the control plane. Our evaluation shows that SafeP4's type system can effectively eliminate common failures in many real-world programs.

- Chapter 5 describes the first algorithm for computing precise ci-specs for network data planes. Our specifications are designed to be efficiently monitorable—concretely, checking that a fixed configuration satisfies a ci-spec can be done in polynomial time. Our algorithm, based on modular program instrumentation, quantifier elimination, and a path-based analysis, is more expressive than prior work, and is applicable to practical network programs. We describe an implementation and show that ci-specs computed by our tool are useful for finding real bugs in real-world data plane programs.

- Chapter 6 describes Avenir, a synthesis tool that automatically generates control-plane operations to ensure uniform behavior across a variety of data planes. Our approach uses counter-example guided inductive synthesis and sketching, adding network-specific optimizations that exploit domain insights to accelerate the search. We prove that Avenir's synthesis algorithm generates correct solutions and always finds a solution, if one exists. We have built a prototype implementation of Avenir using OCaml and Z3 and evaluated its performance on realistic scenarios for the ONOS SDN controller and on a collection of benchmarks that illustrate the cost of retargeting a control plane from one pipeline to another. Our evaluation demonstrates that Avenir can manage data plane heterogeneity with modest overheads

- Chapter 7 proposes a new framework for synchronizing configurable programs, called Relational Hoare Lenses (RHLenses). As the name suggests, this framework can be seen as the marriage of two apparently unrelated lines of work: one focused on the design of bidirectional programming constructs and the other focused on logical tools for reasoning about pairs of programs. The combination of these two approaches is both elegant and powerful. As we will show in this paper, RHLenses neatly generalize prior work on relational program logics, and we believe they can be used to solve practical problems as well.

1.5 Attribution and Acknowledgements

The work presented in this these comprises work from a collection of papers:

1. Chapter 2 and Chapter 3 are based on the following publication, which also appeared in Eichholtz's dissertation *Eichholz, Matthias, Eric Hayden Campbell, Nate Foster, Guido Salvaneschi and Mira Mezini. 2019. How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4. 33rd European Conference on Object-Oriented Programming, ECOOP, (July 2019)*
2. Chapter 4 and Chapter 5 are based on the following publication: *Eric Hayden Campbell, Hossein Hojjat, and Nate Foster. 2024. Computing Precise Control Interface Specifica- tions. Proceedings of the ACM on Programming Languages 8, OOPSLA2(October 2024).*
3. Chapter 6 is based on the following publication: *Eric Hayden Campbell, William T. Hallahan, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soule, and Nate Foster. 2021. 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI (April 2021)*

4. Chapter 7 is based on in-review work with the following attribution: *Eric Hayden Campbell, Mark Barbone, and Nate Foster. 2024. In Review.*

CHAPTER 2

DATA PLANE PROGRAMMING

Luck is a very thin wire between survival and disaster, and not many people can keep their balance on it.

Hunter S. Thompson

Before we can get into verified configuration, we need to develop a working model of data plane programming. As our guide, we'll use the *de facto* industry standard data plane programming language called P4. Rather than model P4's full complexity, we'll focus primarily on the core constructs of the language related to its interface with the control plane.

P4 is a domain-specific language designed for processing packets—i.e., arbitrary sequences of bits that can be divided into (i) a set of pre-determined *headers* that determine how the packet will be forwarded through the network, and (ii) a *payload* that encodes application-level data. P4 is designed to be protocol-independent, which means it handles both packets with standard header formats (e.g., Ethernet, IP, TCP, etc.) as well as packets with custom header formats defined by the programmer. Accordingly, a P4 program first *parses* the headers in the input packet into a typed representation. Next, it uses a *match-action pipeline* to compute a transformation on those headers—e.g., modifying fields, adding headers, or removing them. Finally, a *deparser* serializes the headers back into a packet, which can be output to the next device. A depiction of this abstract forwarding model is shown in Figure 2.1.

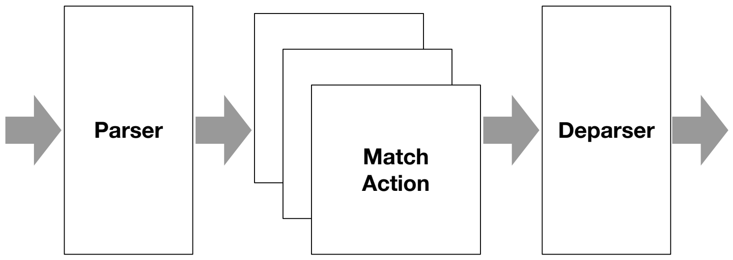


Figure 2.1: Abstract forwarding model.

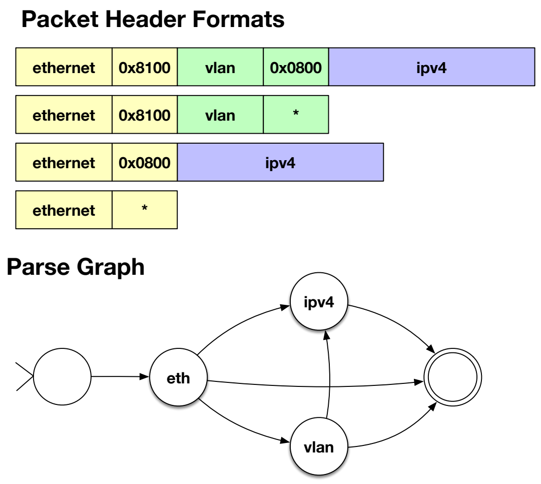
The match-action pipeline relies on a data structure called a *match-action table*, which encodes conditional processing. More specifically, the table first looks up the values being tested against a list of possible entries, and then executes a further snippet of code depending on which entry (if any) matched. However, unlike standard conditionals, the entries in a match-action table are not known at compile-time. Rather, they are inserted and removed at run-time by the control plane, which may be logically centralized (as in a software-defined network), or it may operate as a distributed protocol (as in a conventional network).

The rest of this section describes P4's typed representation, how the parsers, and deparsers convert between packets and this typed representation, and how control flows through the match-action pipeline.

Header Types and Instances Header types specify the internal representation of packet data within a P4 program. For example, the first few lines of the following snippet of code:

```
header ethernet_t {
  dstAddr: bit<48>;
  srcAddr: bit<48>;
  etherType: bit<16>;
}
struct headers {
  ethernet_t ethernet;
  ethernet_t inner_ethernet;
}
```

declare a type (`ethernet_t`) for the Ethernet header with fields `dstAddr`, `srcAddr`,



```
state start {
  return parse_eth;
}
state parse_eth {
  pkt.extract(hdr.ethernet);
  return select(latest.etherType)
     {
  0x8100 : parse_vlan;
  0x0800 : parse_ipv4;
  default: accept;
  }
}
state parse_vlan {
  pkt.extract(hdr.vlan)
  return select(latest.etherType)
     {
    0x0800: parse_ipv4;
    default: accept;
  }
}
state parse_ipv4 {
  pkt.extract(hdr.ipv4);
  return accept;
}
```

Figure 2.2: (Left) Header formats and parse graph that extracts an Ethernet header optionally followed by VLAN and/or IPv4 headers. (Right) P4 code implementing the same parser.

and `etherType`. The type of each field is provided after the colon. While P4 has a wide variety of types from strings, structs, to integers, we'll focus on fixed-width bitvectors, which are the core of the packet-processing logic. Here `bit<`n`>` is the type of a bitvector comprising n bits. The struct called `header` declares two `ethernet_t` instances (`ethernet` and `inner_ethernet`) with global scope. Note that ordinary packets usually have a single Ethernet header, but a tunneling protocol might maintain a second header for encapsulated packets.

Further, each header is equipped with a validity bit that can be read via the method `isValid()`, assigned to true with `setValid()` and `setInvalid()`. The P4 language specification says that when a header h is invalid, the value returned by reading any any field in that header, e.g. $h.f$, is undefined.

```
table forward {
 key = {
  ipv4.isValid() : exact
      ;
  ipv4.isValid() : exact
      ;
  ipv4.dstAddr: ternary;
 }
 actions = {
  nop;
  next_hop;
  remove;
 }
 default_action : nop();
}
```

Runtime Contents of `forward`

Pattern			Action	
ipv4	vlan	ipv4.dstAddr	**Name**	**Data**
1	0	10.0.0.*	next_hop	s, d
0	1	*	remove	

Figure 2.3: P4 tables. `forward` reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance, and calls one of its actions: `nop`, `next_hop`, or `remove`.

Parsers A P4 parser specifies the order in which headers are extracted from the input packet using a simple abstraction based on finite state machines. Extracting into an header instance populates its fields with the requisite bits of the input packet and marks the instance as valid. The code within each state may extract bits from the input packet, modify header instances, conditionally branch, and transition either to another state, to the `ingress` pipeline (indicated by `accept`), which begins the match-action processing, or to the `reject` state which indicates that the packet should be dropped. Figure 2.2 depicts a visual representation of a parse graph for three common headers: Ethernet, VLAN, and IPv4. The instance `ethernet` is extracted first, optionally followed by a `vlan` instance, or an `ipv4` instance, or both.

Tables Interface with the Control Plane The bulk of the processing for each packet in a P4 program is performed using match-action tables that are populated by the control plane. A table (such as the one in Figure 2.3) is defined in terms of (i) the data it reads (indicated by the `key` clause) to determine a matching entry (if any), (ii) the `actions` it may execute, and (iii) an optional `default_action` it executes if no matching entry is found.

The behavior of a table depends on the entries installed at run-time by the control-plane. Each table entry contains a match pattern, an action, and action data. Intuitively, the match pattern specifies the bits that should be used to match values, the action is the name of a pre-defined function (such as the ones in Figure 2.4), and the action data are the arguments to that function. Operationally, to process a packet, a table first scans its entries to locate the first matching entry. If such a matching entry is found, the packet is said to "hit" in the table, and the associated action is executed. Otherwise, if no matching entry is found, the packet is said to "miss" in the table, and the `default_action` (which is a no-op if unspecified) is executed.

A table also specifies the *match-kind* that describes how each header field should match with the patterns provided by the control plane. For the purposes of this dissertation, we only consider `exact` and `ternary`, as ternary matches suffice to implement all other non-`exact` matches. An `exact` match requires the bits in the packet be exactly equivalent to the bits in the controller-installed pattern. A `ternary` match allows wildcards in arbitrary positions, so the controller-installed pattern `0*` would match bit sequences `00` and `01`.

For example, in Figure 2.3, the `forward` table is shown populated with two rules. The first rule tests whether `ipv4` is valid, `vlan` is invalid, and the first 24 bits of `ipv4.srcAddr` equal `10.0.0`, and then applies `next_hop` with arguments s and d (which stand for source and destination addresses). The second rule checks that `ipv4` is invalid, then that `vlan` is valid, and skips evaluating the value of `ipv4.dstAddr` (since it is wildcarded), to finally apply the `remove` action.

```
action next_hop(src, dst) {
 hdr.ethernet.srcAddr = src;
 hdr.ethernet.dstAddr = dst;
 hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

```
action remove() {
 hdr.ethernet.etherType =
    vlan.etherType;
 hdr.vlan.setInvalid();
}
```

Figure 2.4: P4 actions.

Actions are functions containing sequences of primitive commands that perform operations such as adding and removing headers, assigning a value to a field, adding one field to another, etc. For example, Figure 2.4 depicts two actions: the `next_hop` action updates the Ethernet source and destination addresses with action data from the controller; and the `remove` action copies EtherType field from the `vlan` header instance to the `ethernet` header instance and invalidates the `vlan` header.

The `next_hop` action assigns its argument `src`, which is provided by the control plane, to the `srcAddr` field of the `ethernet` instance while `remove` copies the `etherType` field from the `vlan` instance to the `ethernet` instance, and then removes the `vlan` instance. Assignments behave as expected: evaluating e and storing the result in the header field $h.f$—e.g., `next_hop` decrements `ipv4.ttl` by one and saves the result. The `setInvalid()` header method invalidates its header, behaving like a no-op on invalid headers.

Data Plane Control Blocks P4 control blocks use standard control-flow constructs to execute a pipeline of match-action tables. They manage the order and conditions under which each table is executed. Within a control block, the `apply` command executes a table, and conditionals branch on a boolean expression such as the validity of a header instance.

```
apply {
  if(hdr.ipv4.isValid()  hdr.vlan.isValid) { forward.apply(); }
}
```

The above code applies the `forward` table if one of `ipv4` or `vlan` is valid.

Deparser The deparser reassembles the final output packet, after all processing has been done by serializing each valid header instance in some order. The `emit` function serializes the packet into its bits, and then appends it to the outgoing packet, `pkt`. The following code exemplifies code that emits Ethernet, IPv4 and TCP headers, in that order.

```
pkt.emit(hdr.ethernet); pkt.emit(hdr.ipv4); pkt.emit(hdr.tcp);
```

2.1 Data Plane Programs Have Bugs

Having introduced the basic features of P4, we now present five categories of bugs found in open-source programs that arise due to reading and writing invalid headers. There is one category for each of the following syntactic constructs: (1) parsers, (2) controls, (3) table reads, (4) table actions, and (5) default actions.

To identify the bugs we surveyed a benchmark suite of 15 research and industrial P4 programs that are publicly available on GitHub and compile to the BMv2 [82] backend. Later, in Section 6.6, we will report the number of occurrences of each of these categories in our benchmark suite detected by our approach.[1]

2.1.1 Parser Bugs

The first class of errors is due to the parser being too conservative about dropping malformed packets, which increases the set of headers that may be invalid in the control pipeline. In most programs, the parser chooses which headers to `extract`

[1]The examples we use in this chapter are from $P4_{14}$, due to the limited number of $P4_{16}$ programs available when this work was completed. Nonetheless the issues we address also persist in the latest version of the language, $P4_{16}$. We use $P4_{16}$ syntax for modernity.

```
/* UNSAFE */
state parse_ethernet {
 pkt.extract(ethernet);
 transition
   select(ethernet.etherType) {
    0x0800 : parse_ipv4;
    default : accept;
   }
}
state parse_ipv4 {
 pkt.extract(ipv4);
 transition
  select(ipv4.protocol) {
    6 : parse_tcp;
    default : accept;
 }
}
```

```
/* SAFE */
state parse_ethernet {
 extract(ethernet);
 transition
  select(ethernet.etherType) {
    0x0800 : parse_ipv4;
      default : reject;
  }
}
state parse_ipv4 {
 pkt.extract(ipv4);
 transition
   select(ipv4.protocol) {
    6 : parse_tcp;
   default : reject;
  }
}
```

```
state parse_tcp {
  pkt.extract(tcp);
  transition accept;
}
```

```
apply {
  if(tcp.syn == 1 and ...){...}
}
```

Figure 2.5: Left: unsafe code in NetHCF; Right: our type-safe fix; Bottom: common code.

based on the fields of previously-extracted headers using P4's version of a switch statement, `select`. Programmers often fail to handle packets that fall through to the `default` case of these `select` statements.

An example from the NetHCF [120, 8] codebase illustrates this bug. NetHCF is a research tool designed to combat TCP spoofing. As shown in Figure 2.5, the parser handles TCP packets in `parse_ipv4` and redirects all other packets to the `ingress` control. Unfortunately, the `ingress` control (bottom right) does not check whether `tcp` is valid before accessing `tcp.syn` to check whether it is equal to `1`. This is unsafe since `tcp` is not guaranteed to be valid even though it is required to be valid in the `ingress` control.

To fix this bug, we can simply `reject` the failed packets. In general, we may want to define a parser exception, `unsupported`, with a specific handler for unexpected packets, thereby protecting the `ingress` from having to handle unexpected packets. Note however, that this fix might not be the best solution, since it alters

```
/* UNSAFE */
 apply {

 process_cache();
 process_value();

 ipv4_route.apply();
}
```

```
/* SAFE */
apply{
  if(nc_hdr.isValid()) {
    process_cache();
    process_value();
 }
 ipv4_route.apply();
}
```

```
control process_cache {
    check_cache_exist.apply();
    ...
}
```

```
table check_cache_exist {
    key = { nc_hdr.key : exact }
    actions = { ... }
}
```

Figure 2.6: Left: unsafe code in NetCache; Right: our type-safe fix; Bottom: Common code

the original behavior of the program. However, without knowing the programmer's intention, it is generally not possible to automatically repair a program with undefined behavior.

2.1.2 Control Bugs

Another common bug occurs when a table is executed in a context in which the instances referenced by that table are not guaranteed to be valid. This bug can be seen in the open-source code for NetCache [63], a system that uses P4 to implement a load-balancing cache. The parser for NetCache reserves a specific port (`8888`) to handle its special-purpose traffic, a condition that is built into the parser, which extracts `nc_hdr` (i.e., the <u>N</u>et<u>C</u>ache-specific header) only when UDP traffic arrives from port `8888`. Otherwise, it performs standard L2 and L3 routing. Unfortunately, the `ingress` control node (Figure 2.6) tries to access `nc_hdr` before checking that it is valid. Specifically, the `reads` declaration for the `check_cache_exists` table, which is executed first in the `ingress` pipeline, presupposes that `nc_hdr` is valid. The invocation of the `process_value` table (not shown) contains another instance of the same bug.

To fix these bugs, we can wrap the calls to `process_cache` and `process_value` in an conditional that checks the validity of the header `nc_hdr`. This ensures that `nc_hdr` is valid when `process_cache` refers to it.

2.1.3 Table Reads Bugs

A similar bug arises in programs that contain tables that first match on the validity of certain header instances before matching on the fields of those instances. The advantage of this approach is that multiple types of packets can be processed in a single table, which saves memory. However, if implemented incorrectly, this programming pattern can lead to a bug, in which the `reads` declaration matches on bits from a header that may not be valid!

The `switch.p4` program exhibits an exemplar of this bug; it is a "realistic production switch" developed by Barefoot Networks (which has since been purchased by Intel), meant to be used "as-is, or as a starting point for more advanced switches" [68].

An archetypal example of table reads bugs is the `port_vlan_mapping` table of `switch.p4` (Figure 2.7). This table is invoked in a context where it is not known which of the VLAN tags is valid, despite containing references to both `vlan_tag_[0]` and `vlan_tag_[1]` in the `reads` declaration. Adroitly, the programmer has guarded the references to `vlan_tag_[`i`].vid` with keys that test the validity of `vlan_tag_[`i`]`, for $i = 1, 2$. Unfortunately, as written, it is impossible for the control plane to install a rule that will always avoid reading the value of an invalid header. The first match will check whether the `vlan_tag_[0]` instance is invalid, which is safe. However, the very next match will try to read the value of

```
/* UNSAFE */
table port_vlan_mapping {
 key = {
  vlan_tag_[0].isValid() : exact
      ;
  vlan_tag_[0].vid : exact;
  vlan_tag_[1].isValid() : exact
      ;
  vlan_tag_[1].vid : exact;
 } ...
}
```

```
/* SAFE */
table port_vlan_mapping {
 key  = {
  vlan_tag_[0].isValid : exact;
  vlan_tag_[0].vid : ternary;
  vlan_tag_[1].isValid : exact;
  vlan_tag_[1].vid : ternary;
 } ...
}
```

Figure 2.7: Left: a table in `switch.p4` with unprotected conditional reads; Right: our type-safe fix.

the `vlan_tag_[0].vid` field, even when the instance is invalid! This attempt to access an invalid header results in undefined behavior, and is therefore a bug.

It is worthy to note that this code is not actually buggy on some targets—in particular, on targets where invalid headers are initialized with `0`. However, `0`-initialization is not prescribed by the language specification, and therefore this code is not portable across other targets.

The naive solution to fix this bug is to refactor the table into four different tables (one for each combination of validity bits) and then check the validity of each header before the tables are invoked. While this fix is perfectly safe, it can result in a combinatorial blowup in the number of tables, which is clearly undesirable both for efficiency reasons and because it requires modifying the control plane.

Fortunately, rather than factoring the table into four tables, we can replace the `exact` match-kinds with `ternary` match-kinds, which permit matching with wildcards. In particular, the control plane can install rules that match invalid instances using an all-wildcard patterns, which is safe.

In order for this solution to typecheck, we need to assume that the control plane is well-behaved—i.e. that it will install wildcards for the `ternary` matches whenever the header is invalid. In our implementation, we print a warning whenever we make this kind of assumption so that the programmer can confirm that the control plane is well-behaved.

2.1.4 Table Action Bugs

Another prevalent bug arises when distinct actions in a table require different (and possible mutually exclusive) headers to be valid. This can lead to two problems: (i) the control plane can populate the table with unsafe match-action rules, and (ii) there may be no validity checks that we can add to the control to make all of the actions typecheck.

The `fabric_ingress_dst_lkp` table (Figure 2.8) in `switch.p4` provides an example of this misbehavior. The `fabric_ingress_dst_lkp` table reads the value of `fabric_hdr.dstDevice` and then invokes one of several actions: `term_cpu_packet`, `term_fabric_unicast_packet`, or `term_fabric_multicast_packet`. Respectively, these actions require the `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` (respectively) headers to be valid. Unfortunately the validity of these headers is mutually exclusive.[2]

Since `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` are mutually exclusive, there is no single context that makes this table safe. The only facility the table provides to determine which action should be called is `fabric_hdr.dstDevice`. However, the P4 program doesn't establish a relationship between the value of `fabric_hdr.dstDevice` and the validity of any of these three header instances. So, the behavior of this table is only well-defined when

[2]There are other actions in the real `fabric_ingress_dst_lkp`, but these three actions demonstrate the core of the problem.

```
/* UNSAFE */
table fabric_ingress_dst_lkp {
 key = {
  fabric_hdr.dstDevice : exact;
 }


 actions = {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

```
/* SAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
  fabric_hdr_cpu.isValid()
    : exact;
  fabric_hdr_unicast.isValid()
    : exact;
  fabric_hdr_multicast.isValid()
    : exact;
 }
 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

Figure 2.8: Left: unsafe code in `switch.p4`; Right: our type-safe fix.

the input packets are well-formed, an unreasonable expectation for real switches, which may receive *any* sequence of bits "on the wire."

We fix this bug by including validity matches in the `key` declaration, as shown in Figure 2.8. As in Section 2.1.3, this solution avoids combinatorial blowup and extensive control plane refactoring.

In order to type-check this solution, we need to make an assumption about the way the control plane will populate the table. Concretely, if an action a only typechecks if a header h is valid, and h is not necessarily valid when the table is applied, we assume that the control plane will only call a if h is matched as valid. For example, `fabric_hdr_cpu` is not known to be valid when (the fixed version of) `fabric_ingress_dst_lkp` is applied, so we assume that the control plane will only call action `term_cpu_packet` when `fabric_hdr_cpu` is matched as valid. Again, our implementation prints these assumptions as warnings to the programmer, so they can confirm that the control plane will satisfy these assumptions.

```
/* UNSAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }

}
```

```
/* SAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }
 default_action :
    add_value_header_1_act();
}
```

Figure 2.9: Left: unsafe code in NetCache; Right: our type-safe fix.

2.1.5 Default Action Bugs

Finally, *default action* bugs occur when the programmer incorrectly assumes that a table performs some action when a packet misses. The NetCache program (described in Section 2.1.2) exhibits an example of this bug, too. The bug is shown in Figure 2.9, where the table `add_value_header_1` is expected to make the `nc_value_1` header valid, which is done in the `add_value_header_1_act` action. The control plane may refuse to add any rules to the table, which would cause all packets to miss, meaning that the `add_value_header_1_act` action would never be called and `nc_value_1` may not be valid. To fix this error, we simply set the default action for the table to `add_value_header_1_act`, which will force the table to remove the header no matter what rules the controller installs.

The preponderance of real bugs in real P4 programs, for such a simple property, header validity, suggests the need for a lightweight static analysis tool that permits engineers to verify the absence of such bugs. We present such a system in the following chapter,over the course of which the need for careful consideration of the control plane interface will become clear.

CHAPTER 3

STATIC DATA PLANE ANALYSIS AND CONTROL ASSUMPTIONS

Over the past decade, there has been a shift to more flexible platforms in which the functionality of the network is specified in software. Early efforts related to software-defined networking (SDN) [76, 25], focused on the control plane software that computes routes, balances load, and enforces security policies, and modeled the data plane as a simple pipeline operating on a fixed set of packet formats. However, there has been recent interest in allowing the functionality of the data plane itself to be specified as a program—e.g., to implement new protocols, make more efficient use of hardware resources, or even relocate application-level functionality into the network [63, 62]. In particular, the P4 language [20] enables the functionality of a data plane to be programmed in terms of declarative abstractions such as header types, packet parsers, match-action tables, and structured control flow that a compiler maps down to an underlying target device.

Unfortunately, while a number of P4's features were clearly inspired by designs found in modern languages, the central abstraction for representing packet data—header types—lacks basic safety guarantees. To a first approximation, a P4 header type can be thought of as a record with a field for each component of the header. For example, the header type for an IPv4 packet, would have a 4-bit version field, an 8-bit time-to-live field, two 32-bit fields for the source and destination addresses, and so on.

According to the P4 language specification, an instance of a header type may either be valid or invalid: if the instance is valid, then all operations produces a defined value, but if it is invalid, then reading or writing a field yields an undefined result. In practice, programs that manipulate invalid headers can exhibit a variety of faults including dropping the packet when it should be forwarded, or even leaking information from one packet to the next. In addition, such programs are also not portable, since their behavior can vary when executed on different targets.

The choice to model the semantics of header types in an unsafe way was intended to make the language easier to implement on high-speed routers, which often have limited amounts of memory. A typical P4 program might specify behavior for several dozen different protocols, but any particular packet is likely to contain only a small handful of headers. It follows that if the compiler only needs to represent the valid headers at run-time, then memory requirements can be reduced. However, while it may have benefits for language implementers, the design is a disaster for programmers—it repeats Hoare's "mistake," and bakes an unsafe feature deep into the design of a language that has the potential to become the de-facto standard in a multi-billion-dollar industry.

This chapter investigates the design of a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy basic safety guarantees. In particular, this chapter describes SafeP4, a language with a precise semantics and a static type system that can be used to obtain guarantees about the validity of all headers read or written by the program. Although the type system is mostly based on standard features, there are several aspects of its design that stand out. First, to facilitate tracking dependencies between headers—e.g. if the TCP header is valid, then the IPv4 will also be valid—SafeP4 has an expressive algebra of types that tracks validity information at a fine level of granularity. Second, to accommodate the growing collection of extant P4 programs with only modest modifications, SafeP4 uses a path-sensitive type system that incorporates information from conditional statements, forwarding tables, and the control plane to precisely track validity.

To evaluate our design for SafeP4, we formalized the language and its type system in a core calculus and proved the usual progress and preservation theorems. We also implemented the SafeP4 type system in an OCaml prototype, P4Check, and applied it to a suite of open-source programs found on GitHub such as `switch.p4`, a large P4 program that implements the features found in modern data center switches (specifically, it includes over four dozen different switching, routing, and tunneling protocols, as well as multicast, access control lists, among other features). We categorize common failures and, for programs that fail to type-check, identify the root causes and apply repairs to make them well-typed. We find that most programs can be repaired with low effort from programmers, typically by applying a modest number of simple repairs.

Overall, the main contributions of this chapter are as follows:

- We propose SafeP4, a type-safe enhancement of the P4 language that eliminates all errors related to header validity.
- We formalize the syntax and semantics of SafeP4 in a core calculus and prove that the type system is sound.
- We implement our type checker in an OCaml prototype, P4Check.
- We evaluate our type system empirically on over a dozen real-world P4 programs and identify common errors and repairs.

3.1 A “Featherweight” P4

Our primary design goal for SafeP4 is to develop a core calculus that models the main features of P4, while guaranteeing that all data from packet headers is manipulated in a safe and well-defined manner. We draw philosophical inspiration from Featherweight Java [58]—i.e., we model the essential features of P4, but prune away unnecessary complexity. The result is a minimal calculus that is easy to reason about, but can still express a large number of real-world data plane programs. For instance, P4 and SafeP4 both achieve protocol independence by allowing the programmer to specify the types of packet headers and their order in the bit stream. Similarly, SafeP4 mimics P4’s use of tables to interface with the control-plane and decide which actions to execute at run-time.

So what features does SafeP4 prune away? We omit a number of constructs that are secondary to how packets are processed—e.g., `field_list_calculations`, `parser_exceptions`, `counters`, `meters`, `action profiles`, etc. It would be relatively straightforward to add these to the calculus—indeed, most are already handled in our prototype—at the cost of making it more complicated. We also modify or distill several aspects of P4. For instance, P4 separates the parsing phase and the control phase. Rather than unnecessarily complicating the syntax of SafeP4, we allow the syntactic objects that represent parsers and controls to be freely mixed. We make a similar simplification in actions, informally enforcing which primitive commands can be invoked within actions (e.g., field modification, but not conditionals).

Another challenge arises in trying to model core behaviors of P4, in that they each have different type systems and behaviors for evaluating expressions. Our calculus abstracts away expression typing and syntax variants by assuming that we are given a set of constants k that can represent values like `0` or `True`, or operators such as `&&` and `?:`. We also assume that these operators are assigned appropriate (i.e., sound) types. With these features in hand, one can instantiate our type system over arbitrary constants.

Another departure from P4 is related to our `add` command, which presents a complication for our expression types. The analogous `setValid()` header method in P4, which simply modifies the validity bit, without initializing any of the fields. This means that accessing any of the header fields before they have been manually initialized reads a non-deterministic value. Our calculus neatly sidesteps this issue by defining the semantics of the `add(h)` primitive to initialize each of the fields of h to a default value. We assume that along with our type constants there is a function `init` that accepts a header type η and produces a header instance of type η with all fields set to their default value. Note that we could have instead modified our type system to keep track of the definedness of header *fields* as well as their validity. However, for simplicity we choose to focus on header validity.

The portion of our type system that analyzes header validity, requires some way of keeping track of which headers are valid. Naively, we can keep track of a set of which headers are guaranteed to be valid on all program paths, and reject programs that reference headers not in this set. However, this coarse-grained approach would lead to a large number of false positives. For instance, the parser shown in Figure 2.2 parses an `ethernet` header and then either boots to `ingress` or parses an `ipv4` header and then either proceeds to the `ingress` or parses an `vlan` header. Hence, at the `ingress` node, the only header that is guaranteed to be valid is the `ethernet` header. However, it is certainly safe to write an `ingress` program that references the `vlan` header after checking it was valid. To reflect this in the type system we introduce a special construct called valid(h) c_1 else c_2, which executes c_1 if h is valid and c_2 otherwise. When we type check this command, following previous work on occurrence typing [113], we check c_1 with the additional fact that h is valid, and we check c_2 with the additional fact that h is not valid.

Even with this enhancement, this type system would still be overly restrictive. To see why, let us augment the parser from Figure 2.2 with the ability to parse TCP and UDP packets: after parsing the `ipv4` header, the parser can optionally extract the `vlan`, `tcp`, or `udp` header and then boot control flow to ingress. Now suppose that we have a table `tcp_table` that refers to both `ipv4` and `tcp` in its `reads` declaration, and that `tcp_table` is (unsafely) applied immediately in the `ingress`. Because the validity of `tcp` implies the validity of `ipv4`, it should be safe to check the validity of `tcp` and then apply `tcp_table`. However, using the representation of valid headers as a set, we would need to ascertain the validity of `ipv4` and of `tcp`.

To solve this problem, we enrich our type representation to keep track of dependencies between headers. More specifically, rather than representing all headers guaranteed to be valid in a set, we use a finer-grained representation—a set of sets of headers that might be valid at the current program point. For a given header reference to be safe, it must to be a member of all possible sets of headers—i.e., it must be valid on all paths through the program that reach the reference.

Overall, the combination of an expressive language of types and a simple version of occurrence typing allows us to capture header dependencies and statically analyze whether a program satisfies the header validity property.

The final challenge with formally modelling P4 lies in its interface with the control-plane, which populates the tables and provides arguments to the actions. While the control-plane's only methodology for managing switch behavior is to populate the match-action tables with forwarding entries, it is perfectly capable of producing undefined behavior. However, if we assume that the controller is well-intentioned, we can prove the safety of more programs.

In our formalization, to streamline the presentation, we model the control plane as a function $\mathcal{CA}(t, H) = (a_i, \bar{v})$ that takes in a table t and the current headers H and produces the action to call a_i and the (possibly empty) action data arguments $\bar{v}$. We also use a function $\mathcal{CV}(t) = \bar{S}$ that analyzes a table t and produces a list of sets of valid headers $\bar{S}$, one set for each action, that can be safely assumed valid when the entries are populated by the control plane. From the table declaration and the header instances that can be assumed valid, based on the match-kinds, we can derive a list of match key expressions $\bar{e}$ that must be evaluated when the table is invoked. Together, these functions model the run-time interface between the switch and the controller. In order to prove progress and preservation, we assume that $\mathcal{CV}$ and $\mathcal{CA}$ satisfy three simple correctness properties: (1) the control plane can safely install table entries that never read invalid headers, (2) the action data provided by the control plane has the types expected by the action, and (3) the control plane will only assume valid headers for an action that are valid for a given packet.

3.1.1 Syntax

The syntax of SafeP4 is shown in Figure 3.1. To lighten the notation, we write $\bar{x}$ as shorthand for a (possibly empty) sequence $x_1, ..., x_n$.

Commands

c ::=

|	extract(h)	EXTRACTION
|	emit(h)	DEPARSING
|	$\mathsf{c_1; c_2}$	SEQUENCE*
|	if(e) $\mathsf{c_1}$ else $\mathsf{c_2}$	CONDITIONAL
|	valid(h) $\mathsf{c_1}$ else $\mathsf{c_2}$	VALIDITY
|	t.apply()	APPLICATION
|	skip	SKIP
|	add(h)	ADDITION*
|	remove(h)	REMOVAL*
|	h.f = e	MODIFICATION*

Actions

a ::=	$\lambda\bar{\mathsf{x}}.\mathsf{c}$	ACTION

Expressions

e ::=

|	v	VALUES
|	h.f	HEADER FIELD
|	x	VARIABLE
|	$\mathsf{k^n}$	CONSTANT

Declarations

d ::=

|	$\mathsf{t(\bar{h}, \overline{(e, m)}, \bar{a})}$	TABLE
|	$\eta\ \{\overline{\mathsf{f} : \tau}\}$	HEADER TYPE
|	$\mathsf{h} \mapsto \eta$	INSTANTIATION

Match Kinds

$m \in \{\mathsf{exact}, \mathsf{ternary}\}$

Constants

$k \in K$

Program

$\mathcal{P}$ ::= $(\bar{d}, c)$

Values

$v \in V$

Header Types

Θ ::=

|	0	CONTRADICTION
|	1	EMPTY
|	h	INSTANCE
|	$\Theta_1 \cdot \Theta_2$	CONCATENATION
|	$\Theta_1 + \Theta_2$	CHOICE

Action Types

α ::= $\bar{\tau} \rightarrow \Theta$

Expression Types

τ ::= `Bool`
| $\bar{\tau} \rightarrow \tau$
| ...

Figure 3.1: Syntax of SafeP4

A SafeP4 program consists of a sequence of declarations $\bar{d}$ and a command c. The set of declarations includes header types, header instances, and tables. Header type declarations describe the format of individual headers and are defined in terms of a name and a sequence of field declarations. The notation $f : \tau$ indicates that field f has type τ. We let η range over header types. A header instance declaration assigns a name h to a header type η. The map $\mathcal{HT}$ encodes the (global) mapping between header instances and header types. Table declarations $t(\overline{h}, (e, m), \overline{a})$, are defined in terms of a sequence of valid-match header instances $\overline{h}$, a sequence of match-key expressions $\overline{(e, m)}$ read in the table, where e is an expression and m is the match-kind used to match this expression, and a sequence of actions $\bar{a}$. The notation $t.valids$ denotes the valid-match instances, $t.reads$ denotes the expressions, and $t.actions$ denotes the actions.

Actions are written as (uncurried) λ-abstractions. An action $\lambda\bar{x}.\ c$ declares a (possibly empty) sequence of parameters, drawn from a fresh set of names, which are in scope for the command c. The run-time arguments for actions (action data) are provided by the control plane. Note that we artificially restrict the commands that can be called in the body of the action to addition, removal, modification and sequence; these actions are identified with an asterisk in Figure 3.1.

The calculus provides commands for extracting (extract), creating (add), removing (remove), and modifying ($\mathsf{h.f} = \mathsf{e}$) header instances. The emit command is used in the deparser and serializes a header instance back into a bit sequence (emit). The if-statement conditionally executes one of two commands based on the value of a boolean condition. Similarly, the valid-statement branches on the validity of h. Table application commands (t.apply()) are used to invoke a table t in the current state. The skip command is a no-op.

The only built-in expressions in SafeP4 are variables x and header fields, written $h.f$. We let v range over values and assume a collection of n-ary constant operators $k^n \in K$.

For simplicity, we assume that every header referenced in an expression has a corresponding instance declaration. We also assume that header instance names h, header type names η, variable names x, and table names t are drawn from disjoint sets of names H,E,V, and T respectively and that each name is declared only once.

3.1.2 Type System

SafeP4 provides two main kinds of types, basic types τ and header types Θ as shown in Figure 3.1. We assume that the set of basic types includes booleans (for conditionals) as well as tuples and function types (for actions).

A header type Θ represents a set of possible co-valid header instances. The type 0 denotes the empty set. This type arises when there are unsatisfiable assumptions about which headers are valid. The type 1 denotes the singleton denoting the empty set of headers. It describes the type of the initial state of the program. The type h denotes a singleton set, $\{\{h\}\}$—i.e., states where only h is valid. The type $\Theta_1 \cdot \Theta_1$ denotes the set obtained by combining headers from Θ_1 and Θ_2—i.e., a product or concatenation. Finally, the type $\Theta_1 + \Theta_2$ denotes the union of Θ_1 or Θ_2, which intuitively represents an alternative.

The semantics of header types, $[\![\Theta]\!]$, is defined by the equations in Figure 3.2. Intuitively, each subset represents one alternative set of headers that may be valid. For example, the header type $\texttt{eth} \cdot (\texttt{ipv4} + 1)$ denotes the set $\{\{\texttt{eth}, \texttt{ipv4}\}, \{\texttt{eth}\}\}$.

To formulate the typing rules for SafeP4, we also define a set of operations on header types: `Restrict`, `NegRestrict`, `Includes`, `Remove`, and `Empty`. The restrict operator `Restrict` Θ h recursively traverses Θ and keeps only those choices in which h is contained, mapping all others to 0. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h. Dually `NegRestrict` Θ h produces only those choices/subsets where h is invalid. `Includes` Θ h traverses Θ and checks that h is always valid. Semantically this says that h is a member of every element of $[\![\Theta]\!]$. `Remove` Θ h removes h from every path, which means, semantically that it removes h from ever element of $[\![\Theta]\!]$. Finally, `Empty` Θ checks

$$
\begin{aligned}
[\![\Theta]\!] &\subseteq \mathcal{P}(\mathit{Header}) \\
[\![0]\!] &= \{\} \\
[\![1]\!] &= \{\{\}\} \\
[\![h]\!] &= \{\{h\}\} \\
[\![\Theta_1 \cdot \Theta_2]\!] &= [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] \\
[\![\Theta_1 + \Theta_2]\!] &= [\![\Theta_1]\!] \cup [\![\Theta_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}(h, f_i) &= \tau_i && \textit{Field lookup} \\
\mathcal{A}(a) &= \lambda \bar{x} : \bar{\tau}.\ c && \textit{Action lookup} \\
\mathcal{CA}(t, H) &= (a_i, \bar{v}) && \textit{Control-plane actions} \\
\mathcal{CV}(t) &= \bar{S} && \textit{Control-plane validity} \\
\mathcal{H}(e) &= \bar{h} && \textit{Referenced Header instances}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{maskable}(t, e, \mathit{exact}) &\triangleq \mathit{false} \\
\mathsf{maskable}(t, e, \mathit{ternary}) &\triangleq \mathcal{H}(e) \subseteq t.\mathit{valids}
\end{aligned}
$$

Figure 3.2: Semantics of header types (left) and auxiliary functions (right).

whether Θ denotes the empty set. We can lift these operators to operate on sets of headers in the obvious way.

Typing Judgement

The typing judgement has the form $\Gamma \vdash \Theta : c \Rrightarrow \Theta'$, which means that in variable context Γ, if c is executed in the header context Θ, then a header instance type Θ' is assigned. Intuitively, Θ encodes the sets of headers that may be valid when type checking a command. Γ is a standard type environment which maps variables x to type τ. If there exists Θ' such that $\Gamma \vdash \Theta : c \Rrightarrow \Theta'$, we say that c is well-typed in Θ.

The typing rules rely on several auxiliary definitions shown in Figure 3.2. The field type lookup function $\mathcal{F}(h, f_i)$ returns the type assigned to a field f_i in header h by looking it up from the global header type declarations via the header instance declarations. The action lookup function $\mathcal{A}(a)$ returns the action definition $\lambda \bar{x} : \bar{\tau}.\ c$ for action a. Finally, the function $\mathcal{CA}(t, H)$ computes the run-time actions for table t, while $\mathcal{CV}(t)$ computes t's assumptions about validity. Both of these

$$\textsc{T-Zero}\quad \frac{\texttt{Empty}\ \Theta_1}{\Gamma \vdash \Theta_1 : c \Mapsto \Theta_2}$$

$$\textsc{T-Skip}\quad \frac{}{\Gamma \vdash \Theta : \mathsf{skip} \Mapsto \Theta}$$

$$\textsc{T-Seq}\quad \frac{\Gamma \vdash \Theta : c_1 \Mapsto \Theta_1 \qquad \Gamma \vdash \Theta_1 : c_2 \Mapsto \Theta_2}{\Gamma \vdash \Theta : c_1; c_2 \Mapsto \Theta_2}$$

$$\textsc{T-If}\quad \frac{\Gamma; \Theta \vdash e : \mathit{Bool} \qquad \Gamma \vdash \Theta : c_1 \Mapsto \Theta_1 \qquad \Gamma \vdash \Theta : c_2 \Mapsto \Theta_2}{\Gamma \vdash \Theta : \mathsf{if\ (e)\ c_1\ else\ c_2} \Mapsto \Theta_1 + \Theta_2}$$

$$\textsc{T-IfValid}\quad \frac{\Gamma \vdash \texttt{Restrict}\ \Theta\ h : c_1 \Mapsto \Theta_1 \qquad \Gamma \vdash \texttt{NegRestrict}\ \Theta\ h : c_2 \Mapsto \Theta_2}{\Gamma \vdash \Theta : \mathsf{valid(h)\ c_1\ else\ c_2} \Mapsto \Theta_1 + \Theta_2}$$

$$\textsc{T-Mod}\quad \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau_i \qquad \Gamma; \Theta \vdash e : \tau_i}{\Gamma \vdash \Theta : h.f = e \Mapsto \Theta}$$

$$\textsc{T-Extr}\quad \frac{}{\Gamma \vdash \Theta : \mathit{extract}(h) \Mapsto \Theta \cdot h}$$

$$\textsc{T-Emit}\quad \frac{}{\Gamma \vdash \Theta : \mathit{emit}(h) \Mapsto \Theta}$$

$$\textsc{T-Add}\quad \frac{}{\Gamma \vdash \Theta : \mathsf{add(h)} \Mapsto \Theta \cdot h}$$

$$\textsc{T-Rem}\quad \frac{}{\Gamma \vdash \Theta : \mathsf{remove(h)} \Mapsto \texttt{Remove}\ \Theta\ h}$$

$$\textsc{T-Apply}\quad \frac{\begin{gathered}\mathcal{CV}(t) = \bar{S} \\ t.\mathit{actions} = \bar{a} \qquad t.\mathit{reads} = \bar{r} \\ \bar{e} = \{e_j \mid (e_j, m_j) \in \bar{r} \wedge \neg \mathsf{maskable}(t, e_j, m_j)\} \\ \cdot; \Theta \vdash e_j : \tau_j \quad \text{for } e_j \in \bar{e} \\ \texttt{Restrict}\ \Theta\ S_i \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a}\end{gathered}}{\Gamma \vdash \Theta : t.\mathit{apply}() \Mapsto \left(\sum_{a_i \in \bar{a}} \Theta'_i\right)}$$

Figure 3.3: Command typing rules for SafeP4

are assumed to be instantiated by the control plane in a way that satisfies basic correctness properties

The typing rules for commands are presented in Figure 3.3. The rule T-Zero gives a command an arbitrary output type if the input type is empty. It is needed to prove preservation. The rules T-Skip and T-Seq are standard. The rule T-If a path-sensitive union type between the type computed for each branch. The rule T-IfValid is similar, but leverages knowledge about the validity of h. So the true branch c_1 is checked in the context `Restrict` Θ h, and the false branch c_2 is checked in the context `NegRestrict` Θ h. The top-level output type is the union of the resulting output types for c_1 and c_2. The rule T-Mod checks that h is guaranteed to be valid using the `Includes` operator, and uses the auxiliary function $\mathcal{F}$ to obtain the type assigned to $h.f$. Note that the set of valid headers does not change when evaluating an assignment, so the output and input types are identical. The rules T-Extr and T-Add assign header extractions and header additions the type $\Theta \cdot h$, reflecting the fact that h is valid after the command executes. Emitting packet headers does not change the set of valid headers, which is captured by rule T-Emit. The typing rule T-Rem uses the `Remove` operator to remove h from the input type Θ. Finally, the rule T-Apply checks table applications. To understand how it works, let us first consider a simpler, but less precise, typing rule:

$$\frac{\begin{array}{c} t.\mathit{reads} = \bar{e} \qquad \cdot;\Theta \vdash e_i : \tau_i \quad \text{for } e_i \in \bar{e} \\ t.\mathit{actions} = \bar{a} \qquad \cdot;\Theta \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a} \end{array}}{\cdot \vdash \Theta : t.\mathit{apply}() \Rrightarrow \left(\sum \Theta'_i\right)}$$

Intuitively, this rule says that to type check a table application, we check each expression it reads and each of its actions. The final header type is the union of the types computed for the actions. To put it another way, it models table application as a non-deterministic choice between its actions. However, while this rule is sound, it is overly conservative. In particular, it does not model the fact that the control plane often uses header validity bits to control which actions are executed.

Hence, the actual typing rule, T-Apply, is parameterized on a function $\mathcal{CV}(t)$ that models the choices made by the control plane, returning for each action a_i, a set of headers S_i that can be assumed valid when type checking a_i. From the reads declarations of the table declaration, we can derive a subset of the expressions

$$\frac{\Gamma, \bar{x} : \bar{\tau} \vdash \Theta : c \Rrightarrow \Theta'}{\Gamma; \Theta \vdash \lambda\ \bar{x} : \bar{\tau}.c : \bar{\tau} \to \Theta'} \quad \text{(T-ACTION)}$$

Figure 3.4: Action typing rule for SafeP4

$$\text{T-CONST}\ \frac{\texttt{typeof}(k) = \bar{\tau} \to \tau' \qquad \Gamma; \Theta \vdash e_i : \tau_i}{\Gamma; \Theta \vdash k(\bar{e}) : \tau'} \qquad \text{T-VAR}\ \frac{x : \tau \in \Gamma}{\Gamma; \Theta \vdash x : \tau}$$

$$\text{T-FIELD}\ \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau}{\Gamma; \Theta \vdash h.f : \tau}$$

Figure 3.5: Expression typing rules for SafeP4

read by the table—e.g., excluding expressions that can be wildcarded when certain validity bits are false. This is captured by the function $\mathsf{maskable}(t, e, m)$ (defined in Figure 3.2) , which determines whether a reads expression e with match-kind m in table t can be masked using a wild-card. The $\mathsf{maskable}$ function is defined using $\mathcal{H}(e)$, which returns the set of header instances referenced by an expression e.

In the example from Section 2.1.3, if an action a_j is matched by the rule $(0, *, 0, *)$, both S_j and e_j are empty.

The typing judgement for actions (Figure 3.4) is of the form $\Gamma; \Theta \vdash a : \bar{\tau} \to \Theta$, meaning that a has type $\bar{\tau} \to \Theta$ in variable context Γ and header context Θ. Given a variable context Γ and header type Θ, an action $\lambda\bar{x}.\ c$ encodes a function of type $\bar{\tau} \to \Theta'$, so long as the body c is well-typed in the context where Γ is extended with $x_i : \tau_i$ for every i.

The typing rules for expressions are shown in Figure 3.5. Constants are type-checked according to rule T-CONSTANT, as long as each expression that is passed as an argument to the constant k has the type required by the `typeof` function. The rule T-VAR is standard.

3.1.3 Operational Semantics

We now present the small-step operational semantics of SafeP4. We define the operational semantics for commands in terms of four-tuples $\langle I, O, H, c\rangle$, where I is the input bit stream (which is assumed to be infinite for simplicity), O is the output bit stream, H is a map that associates each valid header instance with a records containing the values of each field, and c is the command to be evaluated. The reduction rules are presented in Figure 3.6.

The command $\mathsf{extract}(\mathsf{h})$ evaluates via the rule E-EXTR, which looks up the header type in $\mathcal{HT}$ and then invokes corresponding deserialization function. The deserialized header value v is added to to the map of valid header instances, H. For example, assuming the header type $\eta = \{f : \mathit{bit}\langle 3\rangle;\ g : \mathit{bit}\langle 2\rangle;\}$ has two fields f and g and $I = \mathtt{11000}B$ where B is the rest of the bit stream following, then $\mathit{deserialize}_\eta(I) = (\{f = \mathtt{110};\ g = \mathtt{00};\}, B)$.

The rule E-EMIT serializes a header instance h back into a bit stream. It first looks up the corresponding header type and header value in the header table $\mathcal{HT}$ and the map of valid headers respectively. The header value is then passed to the serialization function for the header type to produce a bit sequence that is appended to the output bit stream. Similarly, we assume that a serialization function is defined for every header type, which takes the bit values of the fields of a header value and concatenates them to produce a single bit sequence. We adopt the semantics of P4 with respect to emitting invalid headers. Emitting an invalid

$$\textsc{E-Extr}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{deserialize}_\eta(I) = (v, I')}{\langle I, O, H, \mathsf{extract(h)}\rangle \to \langle I', O, H[h \mapsto v], \mathsf{skip}\rangle}$$

$$\textsc{E-Emit}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{serialize}_\eta(H(h)) = \bar{B}}{\langle I, O, H, \mathsf{emit(h)}\rangle \to \langle I, O.\bar{B}, H, \mathsf{skip}\rangle}
\qquad
\textsc{E-EmitInvalid}\quad \frac{h \notin \mathit{dom}(H)}{\langle I, O, H, \mathsf{emit(h)}\rangle \to \langle I, O, H, \mathsf{skip}\rangle}$$

$$\textsc{E-IfValidTrue}\quad \frac{h \in \mathit{dom}(H)}{\langle I, O, H, \mathsf{valid(h)\ c_1\ else\ c_2}\rangle \to \langle I, O, H, c_1\rangle}$$

$$\textsc{E-IfValidFalse}\quad \frac{h \notin \mathit{dom}(H)}{\langle I, O, H, \mathsf{valid(h)\ c_1\ else\ c_2}\rangle \to \langle I, O, H, c_2\rangle}$$

$$\textsc{E-Mod}\quad \frac{H(h) = r \qquad r' = \{r\ \mathit{with}\ f = v\}}{\langle I, O, H, \mathsf{h.f = v}\rangle \to \langle I, O, H[h \mapsto r'], \mathsf{skip}\rangle}$$

$$\textsc{E-Apply}\quad \frac{\mathcal{CA}(t, H) = (a_i, \bar{v}) \qquad \mathcal{A}(a_i) = \lambda\bar{x}.c_i}{\langle I, O, H, t.\mathit{apply}()\rangle \to \langle I, O, H, c_i[\bar{v}/\bar{x}]\rangle}$$

$$\textsc{E-Add}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{init}_\eta = v}{\langle I, O, H, \mathit{add}(h)\rangle \to \langle I, O, H[h \mapsto v], \mathsf{skip}\rangle}
\qquad
\textsc{E-AddValid}\quad \frac{h \in \mathit{dom}(H)}{\langle I, O, H, \mathit{add}(h)\rangle \to \langle I, O, H, \mathsf{skip}\rangle}$$

$$\textsc{E-Rem}\quad \frac{}{\langle I, O, H, \mathsf{remove(h)}\rangle \to \langle I, O, H \setminus h, \mathsf{skip}\rangle}$$

Figure 3.6: Selected rules of the operational semantics of SafeP4; the elided rules are standard.

header instance—i.e., a header instance which has not been added or extracted—has no effect on the output bit stream (rule E-EmitInvalid). Notice also that the header remains unchanged in H.

Sequential composition reduces left to right, i.e., the left command needs to

E-Const
$$\frac{[\![k]\!](v_1, ..., v_n) = v}{\langle H, k(v_1, ..., v_n)\rangle \rightarrow v}$$

E-Field
$$\frac{H(h) = \{f_1 : n_1, ..., f_k : n_k\}}{\langle H, h.f_i\rangle \rightarrow n_i}$$

Figure 3.7: Selected rules of the operational semantics for expressions.

be reduced to skip before the right command can be reduced (rule E-Seq). The evaluation of conditionals (rules E-If, E-IfTrue, E-IfFalse) is standard. The rules E-Seq, E-If, E-IfTrue and E-IfFalse are relegated to the companion technical report for brevity. The rules for validity checks (E-IfValidTrue, E-IfValidFalse) step to the true branch if $h \in dom(H)$ and to the false branch otherwise.

Table application commands are evaluated according to rule E-Tapply. We first invoke the control plane function $\mathcal{CA}(t, H)$ to determine an action a_i and action data v. Then we use $\mathcal{A}$ to lookup the definition of a_i, yielding $\lambda\bar{x} : \bar{\tau}.\ c_i$ and step to $c_i[\bar{v}/\bar{x}]$. Note that for simplicity, we model the evaluation of expressions read by the table using the control-plane function $\mathcal{CA}$.

The rule E-Add evaluates addition commands add(h). Similar to header extraction, the $init_\eta()$ function produces a header instance v of type η with all fields set to a default value and extends the map H with $h \mapsto v$. Note that according to E-Add-Exist, if the header instance is already valid, add(h) does nothing. Finally, the rule E-Rem removes the header from the map H. Again, if a header h is already invalid, removing it has no effect.

The semantics for expressions is defined in Figure 3.7, using tuples $\langle H, e\rangle$, where H is the same map used in the semantics of commands and e is the expression to evaluate. The rule E-Field reduces header field expressions to the value stored in

$$\textsc{Ent-Empty}\ \frac{}{\cdot\ \models 1} \qquad \textsc{Ent-Inst}\ \frac{dom(H) = \{h\}}{H \models\ h} \qquad \textsc{Ent-Seq}\ \frac{H_1 \models \Theta_1 \quad H_2 \models \Theta_2}{H_1 \cup H_2 \models \Theta_1 \cdot \Theta_2} \qquad \textsc{Ent-ChoiceL}\ \frac{H \models \Theta_1}{H \models \Theta_1 + \Theta_2}$$

$$\textsc{Ent-ChoiceR}\ \frac{H \models \Theta_2}{H \models \Theta_1 + \Theta_2}$$

Figure 3.8: The *Entailment* relation between header instances and header instance types

the heap H for the respective field. To evaluate constants via the rule E-Const (omitting the obvious congruence rule), we assume that there is an evaluation function for constants $[\![k]\!](\bar{v}) = v$ that is well-behaved—i.e., if $\texttt{typeof}(k) = \bar{\tau} \rightarrow \tau'$ and $\overline{v : \tau}$, then $.;.\ \vdash\ [\![k]\!](\bar{v})\ :\ \tau'$. We use these facts to prove progress and preservation.

3.1.4 Safety of SafeP4

We prove safety in terms of progress and preservation. Both theorems make use of the relation $H \models \Theta$ which intuitively holds if H is described by Θ. The formal definition, as given in Figure 3.8, satisfies $H \models \Theta$ if and only if $dom(H) \in [\![\Theta]\!]$.

We prove type safety via progress and preservation theorems. The respective proofs are mostly straightforward for our system—we highlight only the unusual and nontrivial cases below.

Theorem 3.1.1 (Progress). *If $\cdot \vdash \Theta : c \Rrightarrow \Theta'$ and $H \models \Theta$, then either,*

- *$c = skip$, or*

- $\exists \langle I', O', H', c' \rangle.\ \langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$.

Intuitively, progress says that a well-typed command is fully reduced or can take a step.

Theorem 3.1.2 (Preservation)**.** *If* $\Gamma \vdash \Theta_1 : c \Rrightarrow \Theta_2$ *and* $\langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$*, where* $H \models \Theta_1$*, then* $\exists \Theta'_1, \Theta'_2.\ \Gamma \vdash \Theta'_1 : c \Rrightarrow \Theta'_2$ *where* $H' \models \Theta'_1$ *and* $\Theta'_2 < \Theta_2$*.*

More interestingly, preservation says that if a command c is well-typed with input type Θ_1 and output type Θ_2, and c evaluates to c' in a single step, then there exists an input type Θ'_1 and an output type Θ'_2 that make c' well-typed. To make the inductive proof go through, we also need to prove that Θ'_1 describes the same maps of header instance H as Θ_1, and Θ'_2 is semantically contained in Θ_2. We define syntactic containment to be $\Theta_1 < \Theta_2 \triangleq [\![\Theta_1]\!] \subseteq [\![\Theta_2]\!]$. (These conditions are somewhat reminiscent of conditions found in languages with subtyping.)

Proof. By induction on a derivation of $\Gamma \vdash \Theta_1 : c \Rrightarrow \Theta_2$, with a case analysis on the last rule used. We focus on two of the most interesting cases.

Case T-IfValid: $c = \mathsf{valid(h)}\ \mathsf{c_1}\ \mathsf{else}\ \mathsf{c_2}$ and $\Gamma \vdash \texttt{Restrict}\ \Theta_1\ h : c_1 \Rrightarrow \Theta_{12}$ and $\Gamma \vdash \texttt{NegRestrict}\ \Theta_1\ h : c_2 \Rrightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IfValidTrue and E-IfValidFalse

Subcase E-IfValidTrue: $c' = c_1$ and $h \in dom(H)$ and $H' = H$.

Let $\Theta'_1 = \texttt{Restrict}\ \Theta_1\ h$ and $\Theta'_2 = \Theta_{12}$. We have $\Gamma \vdash \Theta'_1 : c' \Rrightarrow \Theta'_2$ by assumption, we have $H \models \Theta'_1$ by an elided lemma formalizing the relationship between RESTRICT and $(\models)$, and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union.

Subcase E-IFVALIDFALSE: $c' = c_2$ and $h \notin dom(H)$ and $H' = H$.

Symmetric to the previous case.

Case T-APPLY: $c = \mathsf{t.apply}()$ and $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ and $t.actions = \bar{a}$ and $\cdot; \Theta \vdash e_j : \tau_j$ for $e_j \in \bar{e}$ and $\texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \to \Theta_i'$ for $a_i \in \bar{a}$ and $\Theta_2 = \sum(\Theta_i')$

Only one evaluation rule applies to c, E-APPLY. It follows that $\mathcal{CA}(t, H) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda \bar{x}.\ c_i$. By inverting T-ACTION, we have $\Gamma, \bar{x} : \bar{\tau}_i; \vdash \texttt{Restrict}\ \Theta\ S_i : c_i \Rrightarrow \Theta_i'$. By control plane assumption (2), we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau}_i$. By the substitution lemma, we have $\Gamma \vdash \texttt{Restrict}\ \Theta\ S_i : c_i[\bar{v}/\bar{x}] \Rrightarrow \Theta_i'$. Let $\Theta_1' = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta_2' = \Theta_i'$. We have shown that $\Gamma \vdash \Theta_1' : c' \Rrightarrow \Theta_2'$, we have that $H' \models \Theta_1'$ by control plane assumption (3), and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union types. $\square$

3.2 Experience (Evaluation)

We implemented our type system in a tool called P4Check that automatically checks P4 programs and reports violations of the type system presented in Figure 3.3. P4Check uses the front-end of `p4v` [72] and handles the full $P4_{14}$ language.[1] Our key findings, which are reported in detail below, show (i) that our type system finds bugs "in the wild" and (ii) that the programmer effort needed to repair programs to pass our type checker is modest.

[1]We also have an open-source prototype implementation for $P4_{16}$ that handles the most common features of $P4_{16}$ (`https://github.com/cornell-netlab/p4check`).

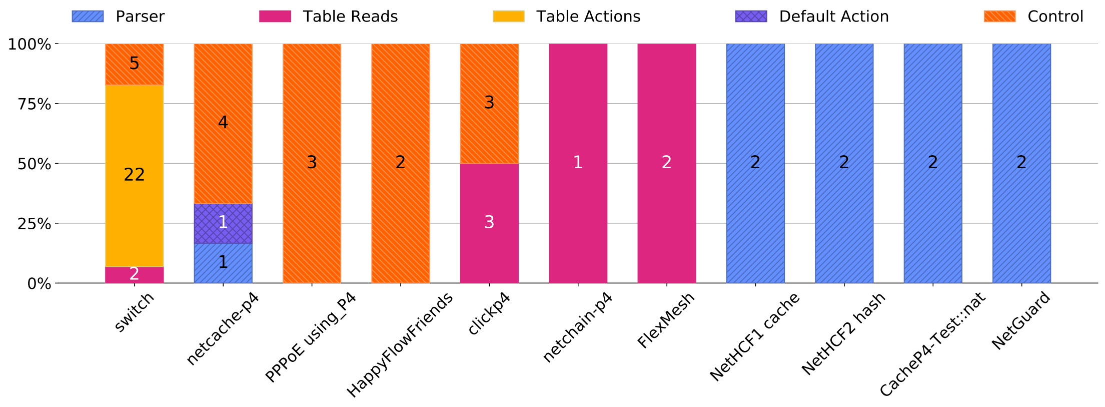


Figure 3.9: Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported at the top of each stacked bar.

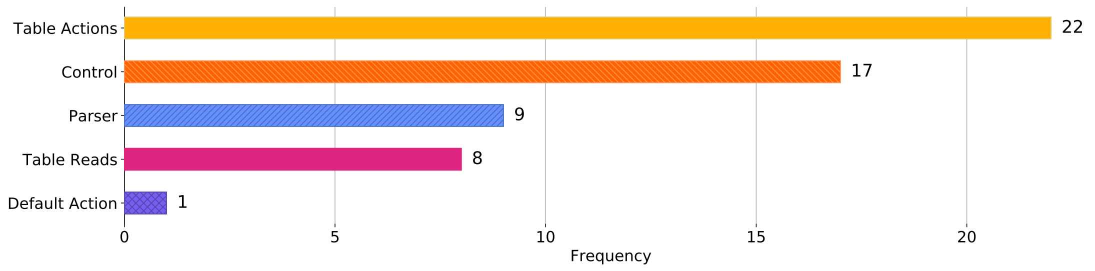


Figure 3.10: Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar

3.2.1 Overview of Bugs in the Wild

We ran P4Check on 15 open source P4_{14} programs[2] of varying sizes and complexity, ranging from 143 to 9060 lines of code. Our criteria for selecting programs was: (1) each program had to be open source, (2) available on GitHub, and (3) compile without errors, (4) and be written either by industrial teams developing production code or by researchers implementing standard or novel network functionality in P4—i.e., we excluded programs primarily used for teaching. Out of the 15 subject

[2]At the time this work was completed, there were significantly more P4_{14} programs than P4_{16} programs available on Github.

programs only 4 passed our type checker, all of which were simple implementations of routers or DDoS mitigation that accepted only a small number of packet types and were relatively small (188–635 lines of code). For the remaining 11 programs (industrial and research) our checker found 418 type checking violations overall.

Frequently, multiple violations produced by P4Check have the same root cause. For example, if a single action `rewrite_ipv4` that rewrites fields `srcAddr` and `dstAddr` for an `ipv4` header is called in a context that cannot prove that `ipv4` is valid, then both references to `ipv4.srcAddr` and `ipv4.dstAddr` will be reported as violations, even though they are due to the same *control* bug (Section 2.1.2)—namely that `rewrite_ipv4` was not called in a context that could prove the validity of `ipv4`. To address this issue, we applied another metric to quantify the number of bugs (inspired by the method proposed by others [67]): we equate the number of bugs in each program with the number of bug *fixes* required to make the program in question pass our type checker. Using this metric, we counted 58 bugs.

We classified the bugs according to the classes described in Section 2.1. Figure 3.9 depicts the per-program breakdown of the frequency of each bug class, and Figure 3.10 depicts the overall frequency of each bug. Notice that even though table action bugs were the most frequent bug (with 22 occurrences), they were only found in a single program (`switch.p4`). These bugs are especially prevalent in this program because of its heavy reliance on correct control-plane configuration. Conversely, there were 9 occurrences across 5 programs for both parser bugs and table reads bugs.

Readers familiar with previous work on `p4v` [72], a recent P4 verification tool, may notice that we detected no default action bugs for the `switch.p4` program, while `p4v` reported many! The reasons for this are two-fold. First, `p4v` allows programmers to verify complex properties, which means that it can express fine-grained conditions on tables and relationships between them. In contrast, we make heuristic assumptions about P4 programs that automatically eliminate many bugs, including some default action bugs. Second, our repairs are often coarse-grained and may enforce a stronger guarantee on the program than may be necessary; using first-order logic annotations, `p4v` programmers manually specify the weakest (and hence more complex) assumptions.

We make no claims about the completeness of our taxonomy. For example, we found one instance, in the HappyFlowFriends program, where the programmer had mistakenly instantiated metadata m as a header, and consequently did not parse m (since metadata is always valid) causing m to (ironically) always be invalid.

3.2.2 P4Check in Action

We reprise the canonical examples of each class of bugs from Section 2.1, describing how P4Check detects them and discussing ways to fix them.

Parser Bugfixes

Recall Figure 2.5, which exhibits the parser bug. The bug occurs because the parser, which extracts IPv4-TCP packets, boots unexpected packets (such as IPv6 or UDP packets) directly to `ingress`, which then assumes that both the `ipv4` and `tcp` headers are valid, even though the parser does not guarantee this fact.

In terms of our type system, the parser produces packets of type $\texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{tcp}))$; however the control only handles packets of type $\texttt{ethernet} \cdot$

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 118, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be
   valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19:error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
./h.p4, line 107, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be
   valid
./h.p4, line 163, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be
   valid
```

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19: error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
```

Figure 3.11: Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet`

`ipv4` · `tcp`. Hence, when typecheck this example, P4Check reports every reference to `tcp` and `ipv4` in the whole program as a violation of the type system. As shown in the top half of Figure 3.11, we get an error message at every reference to `ipv4` or `tcp`. The ubiquity of the reports intimates a mismatch between the parsing and the control types, which gives the programer a hint as how to fix the problem.

When we modify the `default` clause in `parse_ethernet`, as in Figure 2.5, and run our tool again, all of the `ipv4` violations are removed from the output, as shown in the bottom half of Figure 3.11. Then fixing the `parse_ipv4` parser, as in Figure 2.5, causes our tool to output no violations. In particular, the type upon entering the `ingress` control function is `ethernet` · `ipv4` · `tcp`, so all subsequent references to `ipv4` and `tcp` are safe.

```
port.p4, line 248, cols 8-24: warning: assuming either vlan_tag_
    [0] matched as valid or vlan_tag_[0].vid wildcarded

port.p4, line 250, cols 8-24: warning: assuming either vlan_tag_
    [1] matched as valid or vlan_tag_[1].vid wildcarded
```

```
fabric.p4 line 42, cols 41-67: warning: assuming fabric_header_cpu
     matched as valid for rules with action terminate_cpu_packet

fabric.p4, line 57, cols 17-54: warning: assuming
    fabric_header_unicast matched as valid for rules with action
    terminate_fabric_unicast_packet

fabric.p4, line 81, cols 17-56: warning: assuming
    fabric_header_multicast matched as valid for rules with action
    terminate_fabric_multicast_packet
```

Figure 3.12: Warnings printed after fixing `switch.p4`'s reads bug (top), and its actions bug (bottom)

Control Bugfixes

Recall that a control bug occurs when the incoming type presents a choice between two instances that are not handled by subsequent code. The program shown in Figure 2.6 uses a parser that produces the type $\Theta = \texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{udp} \cdot (1 + \texttt{nc\_hdr} \cdot \tau) + \texttt{tcp}))$, where τ is a type for caching operations. Note that `Includes` Θ `nc_hdr` does not hold. However, `process_cache` and `process_value` only type check in contexts where `Includes` Θ `nc_hdr` is true. P4Check reports type violations at every reference to `nc_hdr`. Fixing this error is simply a matter of wrapping the `process_cache()` call in a validity check as demonstrated in Figure 2.6. As NetCache handles TCP and UDP packets as well as its special-purpose packets, we simply apply the IPv4 routing table if the validity check for `nc_hdr` fails.[3]

Table Reads Bugfixes

Table reads errors, as shown in Figure 2.7, occur when a header h is included in the `reads` declaration of a table t with match kind k, and h is not guaranteed to be valid at the call site of t, and if $h \notin$ `valid_reads`(t) or the match-kind of $k \neq$ `ternary`.

In the case of the `port_vlan_mapping` table in Figure 2.7, there is a valid bit for both `vlan_tag_[0]` and `vlan_tag_[1]`, both of which are followed by `exact` matches. To solve this problem, we need to use the `ternary` match-kind instead, which allows the use of wildcard matching. When a field is matched with a wildcard, the table does not attempt to compute the value of the `reads` expression.

This fix assumes that the controller is well behaved and fills the `vlan_tag_[0].vid` with a wildcard whenever `vlan_tag_[0]` is matched as invalid (and similarly for `vlan_tag_[1]`). This also what the SafeP4 type system does, with its maskable checks in the T-Apply rule P4Check prints warnings describing these assumptions to the programmer (top of Figure 3.12), giving them properties against which to check their control plane implementation.

Table Action Bugfixes

Table actions bugs occur when at least one action cannot be safely executed in all scenarios. For example, the table `fabric_ingress_dst_lkp` shown in Figure 2.8 has a table action bug, which can be fixed by modifying the table's `reads` declaration. Recall that the parser will parse exactly one of the headers

[3]Astute readers may detect a parser bug in this example. Hint, the `ipv4_route` table requires `Includes` Θ `ipv4` where Θ is type where it is applied.

`fabric_hdr_cpu`, `fabric_hdr_unicast` and `fabric_hdr_multicast`, which means that when the table is applied at type Θ, exactly one of `Includes` Θ `fabric_hdr_i` for $i \in \{\texttt{cpu}, \texttt{unicast}, \texttt{multicast}\}$ will hold. Now, the action `term_cpu_packet` typechecks only with the (nonempty) type `Restrict` Θ `fabric_hdr_cpu`, and the actions `term_fabric_`i`_packet` only typecheck with the (nonempty) types `Restrict` Θ `term_fabric_`i`_packet` for $i = \texttt{unicast}, \texttt{multicast}$. P4Check suggests that this is the cause of the bug since it reports type violations for all of the references to these three headers in the control paths following from the application of `fabric_ingress_dst_lkp`.

The optimal[4] fix here is to augment the `reads` declaration to include a validity check for each contentious header. We then assume that the controller is well-behaved enough to only call actions when their required headers are valid, allowing us to typecheck each action in the appropriate type restriction. P4Check alerts the programmer whenever it makes such an assumption. We show these warnings for the fixed version of `fabric_ingress_dst_lkp` below the line in Figure 3.12.

Default Action Bugfix

Default action bugs occur when a programmer creates a wrapper table for an action that modifies the type, and forgets to force the table to call that action when the packet misses. The `add_value_header_1` table from Figure 2.9 wraps the action `add_value_header_1_act`, which calls the single line `add_header(nc_value_1)`.

The default action, when left unspecified, is `nop`, which means that if the

[4]Another fix would be to refactor the single into multiple tables, each guarded by a separate validity check. However, combining this kind of logic in a single table helps conserve memory, so in striving to change the behavior of the program as little as possible, we propose modifying the table reads.

pre-application type was Θ, then the post-application type is $\Theta + \Theta \cdot \texttt{nc\_value\_1}$, which does not include `nc_value_1`. Hence, P4Check reports every subsequent reference (on this code path) to `nc_header_1` to be a type violation.

To fix this bug, we need to set the default action to `add_value_1`—this makes the post-application type $\Theta \cdot \texttt{nc\_value\_1} + \Theta \cdot \texttt{nc\_value\_1} = \Theta \cdot \texttt{nc\_value\_1}$, which includes `nc_value_1`, thus allowing the subsequent code to typecheck.

3.2.3 Overhead

It is important to evaluate two kinds of overhead when considering a static type system: overhead on programmers and on the underlying implementation.

Typically, adding a static type system to a dynamic type system requires more work for the programmer—the field of gradual typing is devoted breaking the gargantuan task into smaller commit-sized chunks [24]. Surprisingly, in our experience, migrating real-world P4 code to pass the SafeP4 type system only required modest programmer effort.

To qualitatively evaluate the effort required to change an unsafe program into a safe one using our type system, we manually fixed all of the detected bugs. The programs that had bugs required us to edit between 0.10% and 1.4% of the lines of code. The one exception was PPPoE_using_P4, which was a 143 line program that required 6 line-edits (4%), all of which were validity checks. Conversely, `switch.p4` required 34 line edits, the greatest observed number, but this only accounted for 0.37% of the total lines of code in the program.

Each class of bugs has a simple one-to-two line fix, as described in Section 3.2.2:

adding a validity check, adding a default action, or slightly modifying the parser. Each of these changes was straightforward to identify and simple to make.

Another possible concern is that that extending tables with extra read expressions, or adding run-time validity checks to controls, might impose a heavy cost on implementations, especially on hardware. Although we have not yet performed an extensive study of the impact on compiled code, based on the size and complexity of the annotations we added, we believe the additional cost should be quite low. We were able to compile our fixed version of the `switch.p4` program to the Tofino architecture [59] with only a modest increase in resource usage. Overall, given the large number of potential bugs located by P4Check, we believe the assurance one gains about safety properties by using a static type system makes the costs well worth it.

CHAPTER 4

DATA PLANE VERIFICATION AND THE CONTROL PLANE INTERFACE

While lightweight, SafeP4's static analysis approach described in Chapter 3 is limited by its completeness and its expressiveness. Because header validity often depends on complicated bit-precise invariants about packet data, reasoning precisely about which headers are valid requires dynamic runtime data.

For example, in standard parsers, the Ethernet header's EtherType field determines which Layer 2 header should be parsed next: if its `0x800`, then the IPv4 header is parsed, if its `0x86DD` then the IPv6 header is parsed,... etc. As a consequence, programmers may rely on the value of EtherType to determine whether, say, the IPv4 header is valid. For instance, consider the following code:

```
if (ethernet.etherType == 0x800){
  if (ipv4.ttl > 0){
    ipv4_route.apply();
  }
}
```

which checks whether the EtherType indicates that IPv4 should be valid before running code that relies on the validity of the IPv4 header. Despite the fact that this code is free of invalid header reads, $SafeP4$ will be unable to recognize this fact.

To reason about these dynamic invariants, researchers have employed heavyweight verification. This also lets us naturally extend our concerns beyond the validity of headers.Because P4 is loop-free and finite state, we can employ automated theorem provers to check whether programs satisfy correctness specifications. The `p4v` and Vera papers were the first to show that this was pragmatic and efficient, by compiling P4 programs to Dijkstra's guarded command language (GCL), and

$$
\begin{array}{rcll}
c & \in & \mathrm{GCL} & \\
c & ::= & x := e & \textsc{Assignment} \\
 & \mid & \mathsf{ast}\,\varphi & \textsc{Assertion} \\
 & \mid & \mathsf{asm}\,\varphi & \textsc{Assumption} \\
 & \mid & c; c & \textsc{Sequence} \\
 & \mid & c \,\Box\, c & \textsc{Choice} \\
\end{array}
\qquad x \in \mathsf{Var} \qquad t \in \mathsf{Table}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{[\![c]\!] : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet}) \cup \{\mathsf{error}\}} \\
[\![x := e]\!]\ \ pkt & \triangleq & \{pkt[x \mapsto e(pkt)]\} \\
[\![\mathsf{asm}\,\varphi]\!]\ \ pkt & \triangleq & \{pkt \mid \varphi(pkt) = \mathsf{true}\} \\
[\![\mathsf{ast}\,\varphi]\!]\ \ pkt & \triangleq & \begin{cases} \{pkt\}, & \mathit{if}\ \varphi(pkt) = \mathsf{true} \\ \mathsf{error}, & \mathit{otherwise} \end{cases} \\
[\![c_1; c_2]\!]\ \ pkt & \triangleq & ([\![c_2]\!] \circledcirc [\![c_1]\!])\, pkt \\
[\![c_1 \,\Box\, c_2]\!]\ \ pkt' & \triangleq & [\![c_1]\!]\ \ pkt \uplus [\![c_2]\!]\ \ pkt
\end{array}
$$

Figure 4.1: Syntax (left) and semantics (right) of the Guarded Command Language GCL.

generating corresponding verification conditions. In the next section, we'll outline this approach to verification.

4.1 The Guarded Command Language

In the 70's, Dijkstra invented a simple formalism for reasoning about imperative programs. His *guarded command language* (GCL) had a core set of imperative operators: assignment, assertions, and sequential composition, as well as two non-deterministic control structures: the guarded command, and the do-loop. The upshot of his constructions is that he was able to define *predicate transformers* that defined the semantics of GCL programs in terms of transformations on logical formulae.

In this thesis, we will use a loop-free[1] formulation of GCL (shown in Figure 4.1) that uses a binary nondeterministic operator ($c_1 \,\Box\, c_2$), combined with assert statements ($\mathsf{ast}\ \ \varphi$), and assumptions ($\mathsf{asm}\,\varphi$). This formalization allows the concisely define a compact symbolic compilation function (due to Flanagan & Saxe [46]) that generates logical expressions that precisely capture the semantics of the program.

[1]P4 programs are essentially loop-free

Semantically, we'll use GCL program to denote a nondeterministic function on networking packets, with a possible error state error. What we call a packet pkt is really just a variable valuation from a set of variables Var to set of bitvectors. That is $pkt \in \mathsf{Var} \to 2^*$. We construct the set of all bitvectors 2^*, where $2 = 2^1 = \{\mathsf{true}, \mathsf{false}\}$ and $2^n = 2 \times 2^{n-1}$ for $n > 1$, and $2^* = \bigcup_i 2^i$. To model the nondeterminism, we use sets—that is, the denotation of $c_1 \square c_2$ is—loosely—the union of the denotations of c_1 and c_2.

Formally, the denotation of $c \in \mathrm{GCL}$, is a function $[\![c]\!] : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet}) \cup \{\mathsf{error}\}$. The value of $[\![c]\!]\ pkt$ is either a an error state (see below), or a set $\{pkt_1, \dots, pkt_n\}$ representing all of the possible nondeterministic outputs corresponding to the input pkt. Let's start with our primitives: assignment, assumption, and assertion.

Assigments $x := e$ denote a functional update to the input pkt. The language of expressions e is simply the language of fixed with bitvector arithmetic (Figure 4.2). Lower case variables $x, y, z \subseteq \mathsf{Var}$ range over first-order (bitvector) variables. In the theory of fixed-width bitvectors, each variable is equipped with a bitwidth, which we write $[x]_w$, indicating that values of x must be drawn from 2^w. The expression language defined in the theory of bitvectors could feasibly be any finite function on bitvectors, but typically we take a familiar set of core operations: addition ($+$), subtraction ($-$), multiplication ($*$), division (`div`), shifting ($\ll$, $\gg_a$, $\gg_l$), concatenation ($+\!\!+$), slicing ($\cdot[lo : hi]$), and bitwise operators ($\&$, $|$, $\oplus$, $\dots$). We then write $e(pkt) \in 2^*$ to indicate the evaluation of the expression e on pkt. It's definition is standard, so we omit it. Then, fopr the denotation of $x := e$, we can first evaluate $e(pkt) = v \in 2^*$ and then return the singleton set containing only $\mathsf{update}(pkt, x, v)$, which is the packet that's equivalent to pkt on all variables

$$\begin{array}{rcll} e & ::= & [v]_w & \textsc{Literals} \\ & \mid & [x]_w & \textsc{Variables} \\ & \mid & e \odot e & \textsc{Operations} \\ & \mid & e[lo : hi] & \textsc{Slicing} \\ & \mid & !e & \textsc{Negation} \end{array}$$

$$\odot \in \{+, *, \langle\!\langle, \rangle\!\rangle_a, \rangle\!\rangle_l, +\!\!+, \&, |, \ldots\}$$

$$\begin{array}{rcll} \varphi & ::= & \mathsf{false} & \textsc{Absurd} \\ & \mid & \neg\varphi & \textsc{Negation} \\ & \mid & \varphi \Rightarrow \varphi & \textsc{Implication} \\ & \mid & \varphi \wedge \varphi & \textsc{Conjunction} \\ & \mid & \varphi \vee \varphi & \textsc{Disjunction} \\ & \mid & e \sim e & \textsc{BitVector Comparison} \end{array}$$

$$\sim \;\in \{=, <_s, <_u, \leq_s, \leq_u, >_s, >_u, \geq_s, \geq_u, \ldots\}$$

Figure 4.2: Bitvector Arithmetic: expressions (left) and logical formulae (right)

except x, which is mapped to v.

Next, assumptions, written $\mathsf{asm}\,\varphi$, indicate the assumption that φ holds at the current point in the program. Our formulae φ are derived from the theory of bitvectors, that is the boolean logic defined over signed and unsigned binary bitvector comparison operators ($=$, $<_s$, $<_u$, $>_u$, $>_s$, $\ldots$) over our bitvector arithmetic expressions. For now we'll assume that φ is quantifier-free (this is called the quantifier-free theory of bitvectors, i.e. QFBV). We write $\varphi(pkt) \in \{\mathsf{true}, \mathsf{false}\}$ to indicate evaluation function for formulae—we omit its definition since it is standard. Now, an assumption will check $\varphi(pkt)$ for the input packet pkt. If it is true, the denotation is simply $\{pkt\}$, otherwise, it's the emptyset $\emptyset$.

Conversely, assertions can cause the program to crash. For an assertion $\mathsf{ast}\,\varphi$, if $\varphi(pkt) = \mathsf{false}$, then the denotation is error. Otherwise, the singleton set $\{pkt\}$ is returned. The error state error is different than returning $\emptyset$ because of how it interacts with nondeterministic choice and sequential composition.

Nondeterministic choice, written $c_1 \square c_2$, loosely denotes the union of the denotation of both c_1 and c_2. However, if if either c_1 or c_2 denotes an error state error, then $c_1 \square c_2$ should also denote error. To capture this, we define a "strict" union, written $P_1 \uplus P_2$ where $P_1, P_2 \cup \mathcal{P}(\mathsf{Packet}) \cup \{\mathsf{error}\}$. It is defined as follows:

$$P_1 \uplus P_2 \triangleq \begin{cases} \mathsf{error} & P_1 = \mathsf{error} \textit{ or } P_2 = \mathsf{error} \\ P_1 \cup P_2 & \textit{otherwise} \end{cases}$$

Now, the denotation of $c_1 \square c_2$ for a packet pkt is $[\![c_1]\!]\, pkt \uplus [\![c_2]\!]\, pkt$.

Finally, sequential composition, written $c_1; c_2$ does a similar kind of exception handling. As before, if c_1 or c_2 produces an error, then the whole program is considered to have produced an error. We define the following composition operation:

$$(f_1 \circledcirc f_2)\ pkt \triangleq \begin{cases} \mathsf{error} & \textit{if } f_1(pkt) = \mathsf{error} \\ \biguplus_{pkt' \in f_1(pkt)} f_2(pkt') & \textit{otherwise} \end{cases}$$

Now the denotation of $c_1; c_2$ is simply $[\![c_2]\!] \circledcirc [\![c_2]\!]$.

4.2 Symbolic Compilation

GCL's simplicity pays dividends in the facility with which it can be symbolically compiled. Dijkstra's weakest precondition function $wp(c, \varphi)$ gives the foundational algorithm for computing the weakest assumption ψ on the inputs to c that ensures that running c satisfies ϕ. The formula ψ is called the "weakest precondition". The function is defined below:

$$\begin{aligned} wp(x := e, \varphi) &\triangleq \varphi[x/e] \\ wp(\mathsf{asm}\,\psi, \varphi) &\triangleq \psi \Rightarrow \varphi \\ wp(\mathsf{ast}\,\psi, \varphi) &\triangleq \psi \wedge \varphi \\ wp(c_1; c_2, \varphi) &\triangleq wp(c_2, wp(c_1, \varphi)) \\ wp(c_1 \square c_2, \varphi) &\triangleq wp(c_1, \varphi) \wedge wp(c_2, \varphi) \end{aligned}$$

These predicate transformer semantics can be related directly to our denotational semantics via the following theorem originally due to Dijkstra [32]

Theorem 4.2.1 (Weakest Precondition [32]). *For a program c and a formula φ, with $\psi = wp(c, \varphi)$*

1. *For $pkt, pkt \in \mathsf{Packet}$ s.t. $\psi(pkt) = \mathsf{true}$ and $pkt' \in [\![c]\!]\, pkt$, then $\varphi(pkt')$, and*
2. *For all other ψ' satisfying (1), $\psi' \Rightarrow \psi$.*

With this machinery in hand, we can check that a data plane program satisfies a property so long as we can produce both a GCL program c that models that data plane program, and a corresponding φ that encodes the property we want. We can use high powered solvers to verify the validity of $wp(c, \varphi)$.

4.3 Modeling P4 in GCL

We have all the ingredients for sound model of P4 programs. Most of the operations, assignment, conditionals, parsing, can be modelled either have direct counterparts, or simple encodings into GCL. For instance, we will define $\mathsf{if}(\varphi)\{c_t\}\{c_f\} = (\mathsf{asm}\, \varphi; c_t) \Box (\mathsf{asm}\, \neg\varphi; c_f)$. First we'll enumerate a few of the simpler features, and then address the core question: match-action tables.

4.3.1 Headers

Headers are similar to `structs` in the C language, with typed fields $f_1, \ldots, f_n$, which can be accessed using standard dot notation, e.g. $h.f_i$. Headers are also equipped with a validity bit h.`isValid`() that can be manually manipulated using the `setValid`() and `setInvalid`() methods. We explode headers a list of variables $\mathsf{h\_f}_1, \ldots, \mathsf{h\_f}_n$, one for each field. We also add an explicit validity bit to each header, e.g. $\mathsf{h\_isValid}$. Then `h.setValid`() and h.`setInvalid`() can be modeled as assignment of 0 or 1 to $\mathsf{h\_isValid}$. The validity bit for all headers is initialized to 0.

Metadata is another `struct`-like data representation. They differ from headers only in that they have no validity bit.

4.3.2 Parsing

Parsers are often expressed using a finite state machine abstraction [20], however, because of limitations in programmable data plane hardware [21], these finite state machines are required to terminate within a given bound [29]. In practice, it is straightforward to unroll parser loops.

The `extract` primitive that we focused on in Chapter 3 can be modelled using to the validity bit, and a kind of "havoc" assignment to the extracted fields. For instance, `pkt.extract(h)` where h has a single 8-bit field f can be modeled using the GCL program $\mathsf{h\_isValid} := 1; \mathsf{h\_f} := ?a$ where $?a$ is a fresh variable. Then, any relationships between parser data created by, for instance, parser lookahead, can be expressed in terms of these havoc-variables. In the common case, however, these havoc variables can be eliminated by standard compiler optimizations.

4.3.3 Hash Functions

Hash functions are often used for network functions like heavy-hitter detection [100], or load balancing via equal-cost multipath routing (ECMP). We could model hash functions using uninterpreted functions and concolic execution [97]. However, because they typically occur only once in a pipeline, we can usually get away with modeling them using nondeterminism.

4.3.4 Stateful Operations

Stateful operations are also used to support a variety of applications including in-network telemetry [100], and in-network caching [63], among others. The challenge in programming with state is that stateful `externs` in P4 programs are subject to data races,[2] except when surrounded by the `@atomic` annotation. For simplicity, we treat non-atomic register reads as producing nondeterministic values, while treating registers in `@atomic` blocks like fields in headers.

4.3.5 Match-Action Tables

The key features are the match-action tables. The standard approach [72, 112] is to model tables as a nondeterministic choice between their actions. For instance, the `fabric_ingress_dst_lkp` table from `switch.p4` Figure 2.7 can be modeled as the following GCL program :

$$a_0 \square a_1 \square a_2$$

[2]Section 18.4.1 of the P4 language specification[29]

where a_0 is the GCL encoding of `term_cpu_packet`, a_1 of `term_fabric_unicast_packet`, and a_2 of `term_fabric_unicast_packet`.

This model is certainly sound—if we prove that a property φ holds no matter what combination of actions are executed, then certainly, for any specific configuration of the tables, the data plane program satisfies φ.

However, this approach does yield false alarms. As first noted by the authors of `p4v`, data plane programmers make assumptions about how the control plane will configure tables. In Chapter 5 we'll see that over 50% of a dataset of real programs make nontrivial assumptions about how the tables are programmed, which aligns with previous surveys of p4 programs: `p4v` notes that 9 of its 11 analyzed programs require control plane interfaces, and an analysis of `bf4`'s data suggests that somewhere between 40% and 100% of the programs in its benchmark suite make assumptions about how the control plane programs tables.

We'll address our solution to this problem in Chapter 5, but first, lets look at some prior work

4.4 A First Attempt: Manual Control Interface Specs

The solution that `p4v` offers is manual specification. `p4v` provides programmers with a specification language for specifying the interface to the control plane. This is a good approach to the problem, and has inspired both compositional reasoning (in Π4), and industrial grade verification systems (in Acquila).

In addition to the standard GCL, `p4v` provides users with a simple language of control plane interface specifications. Concretely, the language of expressions is extended with the following atomic predicates, where t ranges over table names, k over table keys, and a over table actions:

$$\mathit{reach}(t) \qquad \mathit{reads}(t,k) \qquad \mathit{hit}(t) \qquad \mathit{miss}(t) \qquad \mathit{action}(t) \qquad \mathit{action\_data}(t,a,x)$$

These predicates are relatively self-explanatory, for instance $\mathit{reach}(t)$ is 1 if program execution is guaranteed to reach table t. Similarly, $\mathit{action}(t)$ returns an identifier indicating the action that table t selects.

The authors suggest making a program-initial assumption that captures the required invariants on the table contents. For instance, one might assume that whenever execution t_1 runs action a, then t_2 must run action b. This assumption can be written as follows:

$$\mathsf{asm}(a = \mathit{action}(t1) \Rightarrow b = \mathit{action}(t2))$$

which is equivalent to $\mathit{action}(t_1) \Rightarrow \mathit{reach}(t_2)$. This interface is straightforward and expressive. Indeed, the authors claim that expert users can write annotations for industrial-grade programs. They do so in their evaluation, and show that, with specifications, they can verify that a suite of industrial grade programs satisfy the header validity property.Unfortunately, this specification language does not provide specifications purely on the table state. For instance, the truth of $\mathit{reach}(t)$ can only be evaluated on the data plane state.

As an improvement upon this, software defined networking engineers at Google have developed a simple annotation language, called `p4-constraints` [107] that they use to eliminate false positives [2]. Their constriants provide a flexible and general framework for expressing custom assumptions about the control plane's configurations. We will call these specifications *control interface specifications* (ci-specs).

In the next chapter, we study how to compute ci-specs. These specifications not only allow verification engineers to build verifiers and fuzzers that rule out false alarms, they also close the loop on a having a precise specification of switch behavior: from just a program denoting an interface and its semantics, to that program, plus an incorporated ci-spec describing the safe use of that interface.

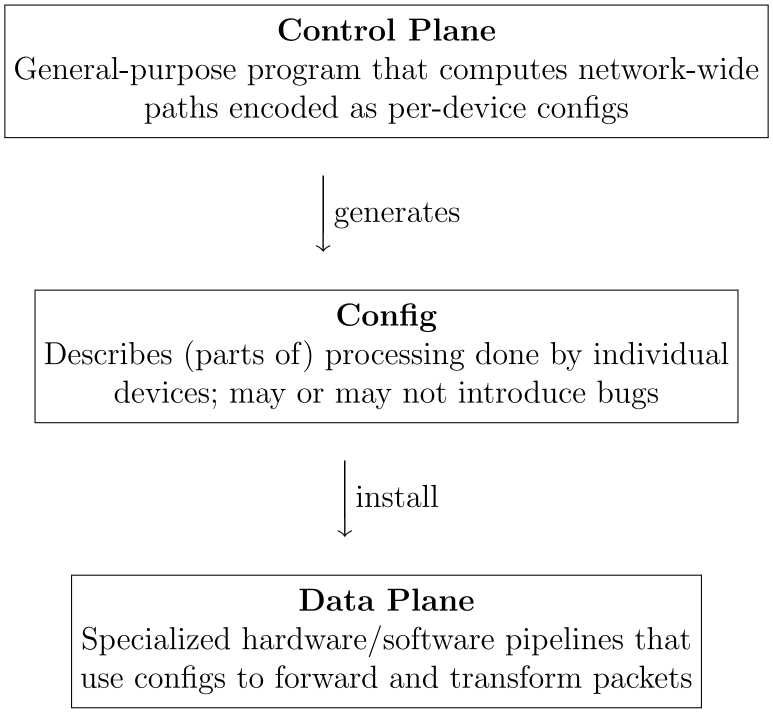
Part II

Control Interface Specifications

CHAPTER 5

COMPUTING PRECISE CONTROL INTERFACE SPECIFICATIONS

Modern networks are increasingly programmable [76, 52, 20, 45]. Abstractly, network architectures can be modeled in terms of two cooperating programs: the *data plane* and the *control plane*. The *control plane* is a general-purpose program that computes forwarding paths through the network topology and generates configurations (configs) for data plane devices such as routers, switches, firewalls, etc. The *data plane* is a collection of restricted (e.g., loop-free and finite-state) programs that process packets efficiently, typically using a pipeline of configurable forwarding tables. This relationship is characterized in the schematic below:


Control Plane
General-purpose program that computes network-wide paths encoded as per-device configs

Config
Describes (parts of) processing done by individual devices; may or may not introduce bugs

Data Plane
Specialized hardware/software pipelines that use configs to forward and transform packets

Figure 5.1: The control plane generates configs that define data plane behavior.

In an ideal world, data plane programs would be written to exhibit correct behavior under any possible config that might be generated by the control plane. However, due to pragmatic hardware-level concerns, programmers make simplifying assumptions about which configs its controller will generate. Unfortunately, existing data plane verification tools take an adversarial perspective, assuming that anything the control plane *can* do it *will* do. Consequently, these tools are subject to false alarms—i.e., configs that violate a given property but will never be generated [72, 2].

To address this problem, the research community has proposed several solutions. VERA uses a runtime monitor that inlines the config and re-verifies the configured data plane program every time the control plane generates a new config [108]. Intel's `p4v` tool and Google's `p4-constraints` library use first-order formulae to specify assumptions about the control plane-generated configs. These constraints are then used to rule out false alarms during verification [72, 2, 97], and to monitor the configs generated by the control plane [107]. However, re-verifying the data plane every time the config changes is expensive, and writing assumptions by hand is complicated and error-prone. How can programmers be certain the control plane will satisfy complex requirements on configs?

Computing Interface Specifications A different approach is to compute a *precise* specification for the interface between the control plane and data plane. We call these descriptions *control interface specifications* (ci-specs). Rather than declaring that a data plane program is "verified" or "unverified", a ci-spec characterizes the conditions that configs must satisfy for the data plane program to satisfy its correctness properties. Hence, it shifts the onus for establishing correctness to the control plane—provided its configs satisfy the ci-spec, the data plane will behave as expected; conversely, if its configs violate the ci-spec, the data plane will be buggy. The ci-specs can be used to monitor the control plane—configs that violate the ci-spec can be logged for offline analysis or rejected outright.

Precise and Efficient Control Interface Specifications In this chapter, we propose Capisce, the first inference engine capable of computing *precise* and *efficiently control-monitorable* ci-specs. Informally, a *precise* ci-spec is both safe, meaning that satisfying configs trigger no bugs, and tight, meaning that violating configs have at least one packet that triggers a bug. Note that computing a *precise* ci-spec has a well-studied solution—we can compute the *weakest precondition* for the data plane and universally quantify over the variables that describe the packet state (Section 5.2, also VERA [108], `p4v` [72]). However, checking that a config satisfies an arbitrary universally-quantified formula is expensive [70]. Instead, Capisce produces ci-specs that the control plane can monitor efficiently. We define a class of *efficiently control-monitorable sentences* (ECMS) and show that every ECMS has *polynomial* complexity. Importantly, Capisce infers *precise* ci-specs in ECMS.

To characterize the complexity of ci-spec inference, we show that it is equivalent to quantifier elimination (QE) in the quantified theory of bitvectors (QBV). In one direction, we describe a compiler pipeline from a high-level model of pipeline programs called the guarded pipeline language (GPL) to the theory of bitvectors with uninterpreted functions (UFBV), and we show how to use QE on specific variables to produce a precise ci-spec in ECMS. In the other direction, we show how to reduce QE to the problem of computing ci-specs—i.e., we produce a simple GPL program whose ci-spec requires eliminating a universal quantifier.

A Practical Implementation Based on Path-Based Heuristics The correspondence between ci-spec inference and QE provides a daunting complexity challenge for the practical tractability of ci-spec inference. In particular, while QE *can* be solved in a finite domain by enumerating the possible instantiations for the quantified variable, a strategy affectionately known as *bit-blasting*, this strategy isn't tractable for real-world data plane programs that manipulate thousands of bits.

For practical programs, however, it is often possible to side-step the worst-case complexity. We draw inspiration from two software engineering folk theorems: (1) "programs are usually correct" and (2) "bugs have simple causes." We interpret (1) to mean that most program *paths* are correct, and the remaining paths are "buggy." Similarly, we interpret (2) to mean that among those relatively few buggy paths, it suffices to compute ci-specs for only a few of *those*.

Capisce leverages these path-based insights in its core algorithm CEGQE: a counterexample-guided inductive inference (CEGIS) loop that uses counterexample paths to iteratively strengthen a candidate ci-spec until it is strong enough to prove the data plane program correct. The precision comes from ensuring that the strengthening step never "overshoots"—i.e., the candidate ci-spec ψ never becomes strictly stronger than the weakest ci-spec.

We have implemented our approach in a tool called Capisce (Section 5.7), and used it to check a standard safety property on a collection of practical programs. Our experiments show that Capisce is able to handle real-world programs, and effectively finds bugs, while only exploring a tiny fraction of these programs' paths (e.g., for our repaired version of `fabric.p4`, only .00000000049%).

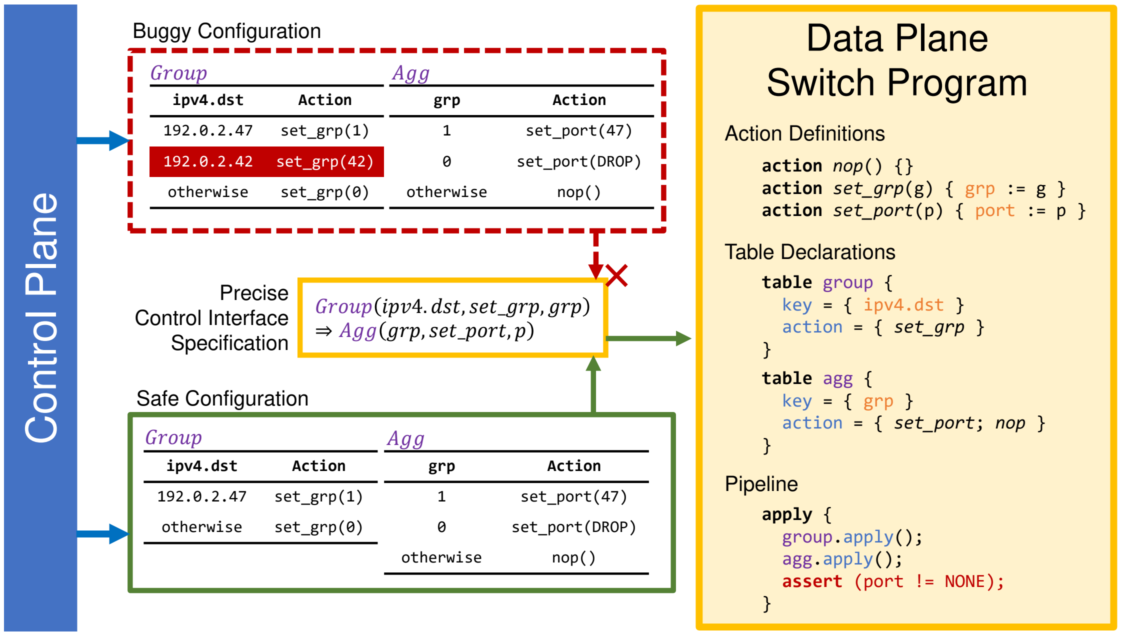
Contributions Overall, this paper makes the following contributions:

- a formal model of data plane pipelines in our new language GPL, and a compiler from GPL to the quantifier-free theory of bitvectors and uninterpreted functions (QFUFBV);
- the class of *efficiently control-monitorable sentences* (ECMS) and a proof that inferring precise ci-specs in this class is *equivalent* to quantifier elimination (QE) in the theory of bitvectors;
- an iterative-strengthening algorithm (CEGQE) for computing precise ci-specs in ECMS that exploits software engineering insights;
- an implementation of Capisce in OCaml, leveraging Princess and Z3 as black-box QE engines;
- an evaluation of Capisce on a benchmark suite of real-world data plane programs, which shows that Capisce can compute precise ci-specs for real-world P4 programs.

5.1 Background and Motivation

In a data plane program, the programmer declares a set of *match-action tables*, and then specifies a conditional *pipeline* that determines the order in which the tables are executed, or *applied*.

A table declaration comprises two components: a *key* and a set of *action*s. The *key* is a list of expressions $e_1, \ldots, e_n$ whose runtime values are used to determine which action is executed. An action is simply a function whose arguments are


Control Plane
Buggy Configuration
Precise Control Interface Specification
Safe Configuration
$Group(ipv4.dst, set\_grp, grp) \Rightarrow Agg(grp, set\_port, p)$

Buggy Configuration

Group

ipv4.dst	Action
192.0.2.47	set_grp(1)
192.0.2.42	set_grp(42)
otherwise	set_grp(0)

Agg

grp	Action
1	set_port(47)
0	set_port(DROP)
otherwise	nop()

Safe Configuration

Group

ipv4.dst	Action
192.0.2.47	set_grp(1)
otherwise	set_grp(0)

Agg

grp	Action
1	set_port(47)
0	set_port(DROP)
otherwise	nop()

Data Plane Switch Program

Action Definitions

```
    action nop() {}
    action set_grp(g) { grp := g }
    action set_port(p) { port := p }
```

Table Declarations

```
    table group {
      key = { ipv4.dst }
      action = { set_grp }
    }
    table agg {
      key = { grp }
      action = { set_port; nop }
    }
```

Pipeline

```
    apply {
      group.apply();
      agg.apply();
      assert (port != NONE);
    }
```

Figure 5.2: An example data plane pipeline program (right) and with an asserted ci-spec (bottom right). Capisce computes a precise ci-spec (center), which ensures that the pipeline satisfies the spec. If the control plane (left) installs a bad config (top), it is rejected. Safe configs, like the one shown on the bottom, are accepted and can be safely installed into the pipeline program.

determined by the table itself—these arguments are called *action data*. As an example, consider the table below:

```
action nop () {}
action set_port (p) { port = p
    }
table fwd {
  key = { ipv4.dst }
  actions = { set_port; nop }
}
```

ipv4.dst	**Action**
192.0.2.47	set_port(47)
192.0.2.42	set_port(42)
otherwise	set_port(DROP)

The table `fwd`, defined in the pseudocode on the left, has a single expression as its key: the variable `ipv4.dst` that holds the IPv4 destination address. It also has two possible actions: `nop` and `set_port`. At runtime, the table's configuration (shown on the right above) will read the value of `ipv4.dst` and run either the `set_port` action, or `nop`.

As defined above, the `nop` action has no action data parameters and executes no operations, while the `set_port` action assigns its single action data parameter `p` to the `port` variable. Whenever a config indicates that the `set_port` action should be run, it must provide an argument, called *action data*, to the `set_port` function.

At runtime, a match action table is a kind of lookup table whose entries are configured by the control plane. To *apply* or *run* a table means evaluating its key expressions, finding the matching table entry, and executing the indicated action with the indicated action data. For example, the table below is to the control plane's config. This table has three entries, or *rows*. The first two execute the `set_port` action with action data n whenever the IPv4 destination address is `192.0.2.`n for $n \in \{42, 47\}$. The final row executes `set_port` with action data `DROP` for every other packet.

These table configs are fundamental to determining the functionality of the switch. To see this, let's look at another example, shown in Figure 5.2. This pipeline exhibits a common pattern known as *link aggregation* [111, 41]. In this program, packet forwarding is divided into two tables: `group`, which computes a forwarding group ID for each packet; and `agg`, which maps each group ID to its forwarding port. In more detail, `group` looks up the IPv4 destination address (`ipv4.dst`) in the controller-provided config, which determines the action to be run. The `group` table only has a single allowed action, `set_group`, which assigns its action data to the `grp` field. For example, first row of the example config shown in Figure 5.2 for `group` assigns the `grp` field to `1` whenever `ipv4.dst` is `192.0.2.47`. Then, `agg` looks up the new `grp` in its config and either runs `nop`, which does nothing, or `set_port`, which assigns its action data `p` to the `port` field. Continuing the example, the first row of the config for `group` sets the port to `47` when `grp` is `1`. Running these configured tables in sequence has the effect of forwarding packets with `ipv4.dst` equal to `192.0.2.47` on port `47`.

The layer of indirection provided by `group` and `agg` is extremely valuable to network operators. Networks must react rapidly to hardware failures or changing service demands by forwarding packets on new routes. Unfortunately, modifying the contents of tables can incur high costs in hardware: due to the way that ternary content addressable memories (TCAMs) work, it can take minutes to process modifications that update thousands of entries [114]. The link-aggregation pattern avoids having to routinely execute minutes-long transactions by rerouting link aggregation groups. If many IP addresses map to the same link aggregation group and the adjacent link goes down, the control plane can reroute traffic for all of those IP addresses by updating a single rule.

The price for efficient reconfigurability is correctness—it is possible for the controller to introduce bugs in this program. Concretely, it can violate the so-called *determined forwarding* safety property, which asserts that every packet has a defined `port` value at the end of the pipeline. This is required because on certain hardware devices [116, 54], failing to assign a port value causes the packet to be forwarded on an *undefined port*. In building large systems of critical infrastructure (like networks), we want to avoid undefined behavior, so we classify such behavior "buggy." One config that produces undefined behavior is shown at the top of Figure 5.2. The `group` table maps address `192.0.2.42` to group 42, which triggers the catch-all rule in `agg` and executes `nop`. Hence, in this config, the forwarding behavior for packets with destination `192.0.2.42` is undefined.

5.1.1 Inference of Control Interface Specifications

Rather than rejecting programs for which the control configs *may* introduce buggy behavior, such as the one in Figure 5.2, we propose computing an interface specification ψ that describes the set of configs that ensure the data plane program p satisfies a given specification φ.

For instance, for the example in Figure 5.2, we want all configs for which the `group` table sets the group field to a value for which `agg` runs `set_port`. We call these restrictions *control interface specifications* (ci-specs). Mathematically, we can specify these specifications using first-order logic.

We can represent each table using a function symbol, *Group* for `group` and *Agg* for `agg`. Each function symbol has an argument for each key, and returns both an identifier that indicates which action will run, and the action's data. For notational elegance, when writing ci-specs, we notate these functions as relations, with the implicit understanding that they also adhere to the requisite functional dependencies and totality constraints. For instance, if we write $\mathit{Agg}(g, a, p)$, the variable d corresponds to an input IPv4 address, then a is the output action identifier (either `nop` or `set_port`), and p is the output port value. Formally, a ci-spec for a pipeline program p is a first-order logic formula over the functions induced by their tables.

Our goal is to compute *precise* ci-specs. A ci-spec ψ is *safe* for a program p and spec φ, if p is guaranteed to satisfy φ for all configs that satisfy ψ. Dually, a ci-spec ψ is *tight* for p and φ, if it is satisfied by every config for which p satisfies φ. To define these notions formally, we stipulate some semantics function $[\![p]\!] : \mathsf{Config} \to \mathsf{Packet} \to \mathsf{Packet}$ (see Section 5.2) that takes in a config $\sigma \in \mathsf{Config}$

and produces a function on packets ($pkt \in \mathsf{Packet}$).

Definition 5.1.1 (Safe ci-spec)**.** Given a pipeline p and specification φ, a ci-spec ψ is *safe* if for every config σ, we have: $\sigma \models \psi \Rightarrow \forall pkt.\, [\![p]\!]^{\sigma}\ pkt \models \varphi$

Definition 5.1.2 (Tight ci-spec)**.** Given a pipeline p and specification φ, a ci-spec ψ is *tight* if for every config σ, we have: $(\forall pkt.\, [\![p]\!]^{\sigma}\ pkt \models \varphi) \Rightarrow \sigma \models \psi$

Finally, we say that a ci-spec is *precise* if it is both safe and tight. For example, the ci-spec shown in the center of Figure 5.2 is precise. Note that a precise ci-spec has the property that for each config that does not satisfy it, there is at least one input that causes the data plane program to violate its spec. Hence, precise ci-specs can also be seen as the *weakest*—i.e., the most-permissive ci-spec.

The overall goal of this paper is to solve the following problem:

Definition 5.1.3 (Problem Statement)**.** For a program p and a spec φ, compute a precise ci-spec ψ.

In what follows, we will show how to produce precise ci-specs; but first, we describe previous work in this area, and elucidate why it doesn't suffice in our domain.

5.1.2 Previous Work

The general problem of synthesizing ci-specs has been studied both in and out of the networking community. The `bf4` tool uses program synthesis to infer single-table *necessary* ci-specs [30, 36], which prohibit no good runs. Formally, a ci-spec ψ is *necessary* for a program p and spec φ, if for every config σ s.t. $\sigma \not\models \varphi$, every input packet causes p to violate φ, that is $[\![p]\!]^{\sigma}\ pkt \not\models \varphi$. If `bf4` cannot infer a necessary constraint that is also sufficient, it reports the program as having *true bugs*. In Section 5.7.6, we compare our approach against `bf4` and find that we infer many more safe ci-specs. As an example, when provided with the example from Figure 5.2, `bf4` computes no ci-spec, because there is no *necessary* single-table ci-spec.

The problem of inferring interface specs (i-specs) has also been studied for general-purpose programs. The MaxSafeSpec algorithm synthesizes the weakest i-spec that is a conjunction of formulae over single function symbols [3]. In our context, this syntactic constraint is analogous to `bf4`'s single-table constraint. The difference here is that MaxSafeSpec computes *sufficient* (or *safe*) i-specs. However, the single-function restriction leads to false alarms when used with data plane programs. Returning to the example, MaxSafeSpec would compute $Agg(\texttt{grp}, a, \texttt{port}) \Rightarrow a = \texttt{set\_port}$, which would reject the sound config at the bottom of Figure 5.2.

So, using current approaches, a data plane engineer seeking to compute ci-specs would need to decide between a potentially-unsafe under-approximation, and an over-approximation, which can lead to false alarms. Capisce threads the needle by computing efficient and precise ci-specs, to provide a safety guarantee while minimizing false alarms.

5.2 Modeling

The remainder of this paper describes Capisce, which computes precise and efficient ci-specs. The first step is to obtain a symbolic model of the data plane. To do

$$
\begin{array}{rcll}
\varphi & ::= & \mathsf{false} & \textsc{Absurd} \\
& \mid & \forall [x]_w .\ \varphi & \textsc{Quantify} \\
& \mid & \varphi \Rightarrow \varphi & \textsc{Implication} \\
& \mid & e \sim e & \textsc{Compare}
\end{array}
$$

$$
\begin{array}{l}
x \in \mathsf{Var}, w, \ell \in \mathbb{N}, F \in \mathsf{Func} \\
\sim \ \ \in \{=, <_s, <_u, \leq_s, \leq_u, \ldots\}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & [v]_w & \textsc{Literals} \\
& \mid & [x]_w & \textsc{Variables} \\
& \mid & e \odot e & \textsc{Binary Ops} \\
& \mid & e[lo : hi] & \textsc{Slicing} \\
& \mid & !e & \textsc{Negation} \\
& \mid & F^{w,\ell}(e) & \textsc{Function}
\end{array}
$$

$$
\odot \in \{+, *, \langle\!\langle, \rangle\!\rangle_l, +\!\!+, \&, |, \ldots\}
$$

Figure 5.3: Bitvector Theories. The syntax of UFBV formulae (left) and expressions (middle). The classification of bitvector theories (above right), depending on whether they allow quantifiers ($\forall$) and/or uninterpreted functions (F). The semantics of bitvector expressions are standard.

this, we describe a symbolic compilation pipeline from an abstract model of data planes (Section 5.2.2) to the theory of bitvectors and uninterpreted functions (Section 5.2.1). Our abstract pipeline language (Section 5.2.2) is called the *guarded pipeline language* (GPL), which lets us reason about branching pipelines of tables. We show we can model pipelines as programs in the assume-variant [46] of Dijkstra's guarded command language [32] by leveraging uninterpreted functions (Section 5.2.2). This modeling lets us employ fairly standard symbolic compilation techniques (Section 5.2.4) to develop a symbolic model. We use this symbolic model to compute precise and efficiently monitorable ci-specs.

5.2.1 Theories of Fixed-width Bitvectors

The core theory of bitvectors remains as defined in Section 4.1. However, we extend the logic with function symbols and quantifiers. Our function symbols also have types $2^w \to 2^\ell$ for $w, \ell \in \mathbb{N}$, in the grammar, we write this as $F^{w,\ell}$, but in practice, as with bitwidths elsewhere, we omit these annotations. Lower case greek symbols φ, ψ, χ range over bitvector formulae in UFBV.

The semantics is largely standard, except for its use of configs $\sigma \in \mathsf{Config}$. The set of configs ($\mathsf{Config}$) is the set of functions with type $\mathsf{Func} \to 2^* \to 2^*$. For convenience, we restrict (wlog[1]) the co-domain of σ to be functions from 2^* to 2^*. However, because each $F^{w,\ell}$ has type $2^w \to 2^\ell$, it must be that $\sigma(F) : 2^w \to 2^\ell$. Intuitively, for a function symbol $F \in \mathsf{Func}$, we have that $\sigma(F)$ is a function definition for F. In this sense, configs σ can be viewed as finite sets modeling first-order logic formulae. In addition to configs, we need to define the runtime packet $\mathit{pkt} \in \mathsf{Packet}$. A packet is a valuation function $\mathit{pkt} : \mathsf{Var} \to 2^*$ on variables. We stipulate a standard evaluation function for expressions $\mathcal{E}\left[\!\left[e\right]\!\right]^\sigma \mathit{pkt} = v$ and a satisfaction relation for formulae $\sigma \models_{\mathit{pkt}} \varphi$.

Notice that our theories differs along two dimensions, the language of formulae (φ), and the language of expressions (e). To indicate that a formula φ is syntactically valid in theory $\mathcal{T}$, we write $\varphi \in \mathsf{Form}\mathcal{T}$. We also write $e \in \mathsf{Expr}(\mathcal{T})$, when e is in $\mathcal{T}$'s language of expressions. This is summarized in the following table:

φ	w/ $\forall$	w/o $\forall$
w/ F	UFBV	QBV
w/o F	QFUFBV	QFBV

5.2.2 Syntax and Semantics of the Guarded Pipeline Language (GPL)

This section presents our modeling language for tables, GPL($\mathcal{T}$). The language is parametric over the bitvector theory used in expressions and assumptions. By default, we will assume $\mathcal{T}$ is QFBV (i.e., no quantifiers or uninterpreted functions),

[1] $F(x, y) = \langle [p]_m, [q]_n \rangle$ can be seen as syntactic sugar for $F(x +\!\!+ y)[0 : m] = [p]_m \wedge F(x +\!\!+ y)[m+1 : m+n+1] = [q]_w$

$$
\begin{array}{rcllr}
p & \in & \mathrm{GPL}(\mathcal{T}) & & [\![p]\!]^\sigma : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet}) \\
p & ::= & x := e & \textsc{Assignment} & [\![x := e]\!]^\sigma \; pkt \triangleq \; \{pkt[x \mapsto \mathcal{E}\,[\![e]\!]^\sigma \; pkt]\} \\
 & | & \mathsf{asm}\,\varphi & \textsc{Assumption} & [\![\mathsf{asm}\,\varphi]\!]^\sigma \; pkt \triangleq \; \{pkt \mid \sigma \models_{pkt} \varphi\} \\
 & | & t(e) & \textsc{Table} & [\![t(e)]\!]^\sigma \; pkt \triangleq \; [\![a_i(d)]\!]^\sigma \; pkt \\
 & | & p; p & \textsc{Sequence} & \textit{where } t : 2^n \to \{a_1, \ldots, a_i, \ldots, a_n\} \\
 & | & p \,\square\, p & \textsc{Choice} & \textit{and } \langle i, d\rangle = \sigma(t)(\mathcal{E}\,[\![e]\!]^\sigma \; pkt) \\
 & & x \in \mathsf{Var} & t \in \mathsf{Table} & [\![p_1; p_2]\!]^\sigma \; pkt \triangleq \; \bigcup_{pkt' \in [\![p_1]\!]^\sigma pkt} [\![p_2]\!]^\sigma pkt \\
 & & e \in \mathsf{Expr}(\mathcal{T}) & \varphi \in \mathsf{Form}\mathcal{T} & [\![p_1 \,\square\, p_2]\!]^\sigma \; pkt \triangleq \; [\![p_1]\!]^\sigma \; pkt \cup [\![p_2]\!]^\sigma \; pkt
\end{array}
$$

Figure 5.4: Syntax (left) and semantics (right) of Guarded Pipeline Language GPL($\mathcal{T}$) over a bitvector theory $\mathcal{T}$. Highlighted variants only occur in GPL($\mathcal{T}$); the other variants are Guarded Command Language GCL($\mathcal{T}$).

and will write GPL to denote GPL(QFBV).

The syntax and semantics of GPL($\mathcal{T}$) are presented in Figure 5.4. A GPL($\mathcal{T}$) program is mostly standard comprising: assignment $[x]_w := e$ which assigns $e \in \mathsf{Expr}(\mathcal{T})$ to the w-bit variable x; assumption ($\mathsf{asm}\,\varphi$) which assumes the truth of $\varphi \in \mathsf{Form}\mathcal{T}$; sequential composition ($c; c$); and finally nondeterministic choice ($c \square c$).

The main non-standard constructs found in GPL($\mathcal{T}$) are table declarations and table applications. A table declaration $T = \langle t, n, \mathbf{a}\rangle$ is a tuple comprising a table name variable $t \in \mathsf{Table} \subseteq \mathsf{Func}$, a natural bitwidth indicating the size of its key domain, $n \in \mathbb{N}$, and a set of possible actions $\mathbf{a} \subseteq \mathsf{Action}$. An action $a = \lambda d : w.\ p$ is a function parameterized on a variable d of bitwidth w and runs a straight-line program p that may read d (a *straight-line program* never uses nondeterministic choice). For an argument $[v]_w$, we write $a(v)$ to mean the substitution $p[d \mapsto v]$. When $w = 0$ we use the syntactic sugar $\lambda().\,p$. We use 2^n to refer to the set of bitvectors of width n. To evoke tables' functionality, we stylize their declarations as follows: $t : 2^n \to \mathbf{a}$.

For instance, below we declare the *Agg* and *Group* tables from Figure 5.2, first defining the actions, and then declaring the tables. The key width for the *Group* table is 32 as it reads the IPv4 destination address, a 32-bit field. The *Agg* table reads the grp field, which is also 32 bits.

$$\mathsf{set\_group}\ g \triangleq \mathsf{grp} := g \qquad \mathsf{set\_port}\ p \triangleq \mathsf{port} := p \qquad \mathsf{nop}() \triangleq \mathsf{asm}\,\mathsf{true}$$

$$\mathit{Group} : 2^{32} \to \{\mathsf{set\_group}\} \qquad \mathit{Agg} : 2^{32} \to \{\mathsf{set\_port}, \mathsf{nop}\}$$

A table application is written $t(e)$ for some declared table $t : 2^n \to \mathbf{a}$ and expression $e \in \text{QFUFBV}$ of width n. This variant is highlighted in Figure 5.4. To indicate that a program p may reference a set of declarations $\mathbf{T}$, we write the stylized pair $p[\mathbf{T}]$. With the above definitions, the link aggregation example from Figure 5.2 is written as follows:

$$\mathit{Group}(\mathsf{ipv4.dst}); \mathit{Agg}(\mathsf{grp})$$

Semantically, a GPL($\mathcal{T}$) program p takes in a config $\sigma \in \mathsf{Config}$ and returns a function from packets (Packet) to sets of packets ($\mathcal{P}(\mathsf{Packet})$). Formally, we have a function $[\![p]\!]^{\sigma} : \mathsf{Packet} \to \mathcal{P}(\mathsf{Packet})$, whose semantics are provided in Figure 5.4. Assignment $x := e$ uses the *pkt* and σ to evaluate e to a bitvector v, returning a singleton set containing the packet $pkt[x \mapsto v]$. The notation $pkt[x \mapsto v]$ indicates the packet that is identical to *pkt* except on variable x, which is mapped to v. Next, assumptions ($\mathsf{asm}\,\varphi$) evaluate whether *pkt* satisfies φ in σ: if so, it returns the singleton packet set $\{pkt\}$, otherwise it returns the empty set $\emptyset$. Sequential composition ($p_1; p_2$) is the composition of the denotation of c_1 composed with the denotation of p_2 lifted to sets in the natural way. Similarly, the semantics of nondeterministic choice ($p_1 \,\square\, p_2$) is the union of the denotations of the disjuncts. Again, GPL($\mathcal{T}$)'s most novel construct is table application, $t(x)$, which, semantically, looks up t in the config σ. Then, $\sigma(t)$ returns a pair $\langle i, d\rangle$ of an action identifier i and action data d. The semantics then select the ith action a_i, and run it with its argument.

From this model, we can also define syntactic sugar for trivial and conditional statements. The trivial statement skip does nothing. Conditionals are encoded in the standard way using a combination of assumes and nondeterministic choice:

$$\mathsf{skip} \triangleq \mathsf{asm\,true} \qquad\qquad \mathsf{if}(b)\{c_t\}\{c_f\} \triangleq \mathsf{asm}\, b; c_t \,\square\, \mathsf{asm}\, \neg b; c_f$$

For a formula $\varphi \in \mathsf{Form}\mathcal{T}$, construing $[\![p]\!]^{\sigma}$ to be a relation lets us write $[\![p]\!]^{\sigma} \models \varphi$ to indicate that p satisfies φ under config σ. We find it more evocative to write this as $p[\sigma] \models \varphi$. We also define $p \models \varphi$ to be $\forall \sigma.\ p[\sigma] \models \varphi$.

An aside on types. GPL requires a type system to keep track of bitwidths and ensure they are used consistently throughout a program. However, we will elide this detail as it is standard and unsurprising. We will also omit bitwidths in examples when they are obvious or irrelevant.

5.2.3 Modeling Tables as Uninterpreted Functions

In this section, we show how we can model GPL($\mathcal{T}$)'s tables using uninterpreted functions. We do so by defining a restriction of GPL($\mathcal{T}$) that corresponds to Dijkstra's guarded command language (GCL($\mathcal{T}$)), and then defining a mapping from GPL($\mathcal{T}$) to GCL($\mathcal{T}$), and proving equivalence.

Formally, GCL($\mathcal{T}$) is the subset of GPL($\mathcal{T}$) that excludes table application. Just as we've been letting p range over GPL($\mathcal{T}$) programs, let c range over

GCL($\mathcal{T}$) programs. For GCL($\mathcal{T}$), the default theory is QFUFBV, so we take the convention that GCL indicates GCL(QFUFBV).

We can define $\mathsf{model} : \text{GPL} \to \text{GCL}$ for an application of table $t : 2^w \to \mathbf{a}$. Intuitively, $\mathsf{model}(t(x))$ treats t as an uninterpreted function, and applies it to the key x to produce an action and argument and then runs that action with its argument.

$$\mathsf{model}(t(x)) \triangleq \langle i, d\rangle := t(x); \mathsf{run}_{\mathbf{a}}(i, d)$$

where $\mathsf{run}_{\mathbf{a}}(i, d)$ selects the ith action from the set $\mathbf{a} = \{a_0, \ldots, a_n\}$ and runs it with argument d:

$$\mathsf{run}_{a_0,\ldots,a_n}(i, d) \triangleq \mathsf{asm}\, i = 0; a_0(d) \,\square \cdots \square\, \mathsf{asm}\, i = n; a_n(d)$$

As an example, consider the pipeline from Figure 5.2. We recapitulate its definition in GPL below and show its translation into GCL:

Action Definitions

$\mathsf{set\_group} \triangleq \lambda g.\ \mathsf{grp} := g$

$\mathsf{set\_port} \triangleq \lambda p.\ \mathsf{port} := p$

$\mathsf{nop} \triangleq \lambda().\ \mathsf{asm\,true}$

Table Definitions

$\mathit{Group} : 2^{32} \to \{\mathsf{set\_group}\}$

$\mathit{Agg} : 2^{32} \to \{\mathsf{set\_port}, \mathsf{nop}\}$

GPL pipeline

$\mathit{Group}(\mathsf{ipv4.dst});$

$\mathit{Agg}(\mathsf{grp})$

$$\overset{\mathsf{model}}{\mapsto}$$

GCL Model

$\langle a, g\rangle := \mathit{Group}(\mathsf{ipv4.dst});$

$\mathsf{grp} := g;$

$\langle b, p\rangle := \mathit{Agg}(\mathsf{grp});$

$\mathsf{if}(b = \mathsf{set\_port})\{$

$\quad \mathsf{port} := p$

$\}\{$// *else* $b = \mathsf{nop}$

$\quad \mathsf{skip}$

$\}$

Observe that both tables *Group* and *Agg* have been replaced by function calls that compute output variables a and d. After *Group* is called, we can ignore a

since *Group* only has one action, and simply assign d to grp. Then we run the *Agg* function to compute b and p. We then inspect b to determine which action should be run. If b indicates the $\mathsf{set\_port}$ action, then port is assigned the action data value p, otherwise, b is nop and nothing happens.

We prove that this translation is semantics-preserving.

Theorem 5.2.1 (Adequacy). $[\![p]\!]^\sigma = [\![\mathsf{model}(p)]\!]^\sigma$

Proof. By induction on p. Let $p = t(x)$, as the remaining cases are immediate or by IHs. Let $\mathbf{a} = \{a_0, \ldots, a_n\}$, and $\langle j, d\rangle = \sigma(t)$.

$$\begin{aligned}
[\![\mathsf{model}(t(x))]\!]^\sigma &= [\![\langle i, d\rangle := t(x); \mathsf{run}_{\mathbf{a}}(i, d)]\!]^\sigma \\
&= [\![\mathsf{run}_{\mathbf{a}}(j, d)]\!]^\sigma \\
&= [\![\mathsf{asm}\, i = 0; a_0(d) \,\square \cdots \square\, \mathsf{asm}\, i = n; a_n(d)]\!]^\sigma \\
&= [\![a_j(d)]\!]^\sigma \\
&= [\![t(x)]\!]^\sigma
\end{aligned}$$

□

Our model is the first to precisely characterize the semantics of tables in a logical formalism [72, 112, 108]. We will use it to generate precise symbolic representations of GPL programs.

5.2.4 Symbolic Compilation

By Theorem 5.2.1, to generate a symbolic model of $p \in \mathrm{GPL}$, we need only compile its GCL model $c = \mathsf{model}(p)$. We rely heavily on previous work [46, 32] to produce our symbolic compiler. Our first step is to normalize programs into the passive form [46]. A program is *passive* if it does not have any assignments. We can *passify* a program c by replacing assignments with assumes. Doing so requires minting a new variable index each time a variable is written, and doing some careful bookkeeping to ensure that indices are synchronized across join points. The function $\mathsf{passify} : \mathrm{GCL}(\mathcal{T}) \times \mathbb{N}^{\mathsf{Var}} \to \mathrm{GCL}(\mathcal{T}) \times \mathbb{N}^{\mathsf{Var}}$, takes in two arguments, a GCL program c and a map $\mathcal{I}$ from variables to indices. It returns a passive c' and a map $\mathcal{I}'$ holding the maximum index for each variable. We define $\mathsf{passify}$ below:

$$
\begin{array}{lcl}
\mathsf{passify}(x := e, \mathcal{I}) & \triangleq & \textbf{let } \mathcal{J} \textbf{ be } \mathcal{I}[x \mapsto \mathcal{I}(x) + 1] \textbf{ in} \\
 & & (\mathsf{asm}\, x_{\mathcal{J}(x)} = \mathsf{subst}(\mathcal{I}, e), \mathcal{J}) \\
\mathsf{passify}(\mathsf{asm}\, \varphi, \mathcal{I}) & \triangleq & (\mathsf{asm}\, \mathcal{I}(\varphi), \mathcal{I}) \\
\mathsf{passify}(c_1; c_2, \mathcal{I}) & \triangleq & \textbf{let } c_1', \mathcal{I}_1 \textbf{ be } \mathsf{passify}(c_1, \mathcal{I}) \textbf{ in} \\
 & & \textbf{let } c_2', \mathcal{I}_2 \textbf{ be } \mathsf{passify}(c_2, \mathcal{I}_1) \textbf{ in} \\
 & & (c_1'; c_2', \mathcal{I}_2) \\
\mathsf{passify}(c_1 \,\square\, c_2, \mathcal{I}) & \triangleq & \textbf{let } c_1', \mathcal{I}_1 \textbf{ be } \mathsf{passify}(c_1, \mathcal{I}) \textbf{ in} \\
 & & \textbf{let } c_2', \mathcal{I}_2 \textbf{ be } \mathsf{passify}(c_2, \mathcal{I}) \textbf{ in} \\
 & & \textbf{let } r_1, r_2, \mathcal{J} \textbf{ be } \mathsf{merge}(\mathcal{I}_1, \mathcal{I}_2) \textbf{ in} \\
 & & (c_1'; r_1 \,\square\, c_2'; r_2, \mathcal{J})
\end{array}
$$

where $\mathcal{I} : \mathsf{Var} \to \mathbb{N}$ is a map from variables to natural indices. We define $\mathcal{Z}$ to be the map that indexes each variable with 0. We always initialize $\mathsf{passify}$ with $\mathcal{Z}$. In the above function, each time we see an assignment $x := e$, we rename e according to the current set of indices using a substitution function $\mathsf{subst}(e, \mathcal{I})$, which returns an expression e whose variables have been indexed according to $\mathcal{I}$. We then increment the index for x. Translating assumptions ($\mathsf{asm}\, \varphi$) is similar, we annotate all the variables in φ with their current indices, written $\mathsf{subst}(\varphi, \mathcal{I})$. The sequence case is homomorphic: after passifying c_1 we passify c_2 with the updated indices from c_1.

The hard case is passifying choice ($c_1 \,\square\, c_2$), where we add so-called *residuals*

r_1 and r_2 to each passified program disjunct (c_1' and c_2' above). These residuals are computed by $\mathsf{merge} : \mathbb{N}^{\mathsf{Var}} \times \mathbb{N}^{\mathsf{Var}} \to \text{GCL} \times \text{GCL} \times \mathbb{N}^{\mathsf{Var}}$ which takes in the indexing functions $\mathcal{I}_1$ and $\mathcal{I}_2$ that result from passifying c_1 and c_2 and returns so-called *residuals* r_1 and r_2. The residuals synchronize the indices between c_1' and c_2'. The residual r_1 finds the variables that have a lower maximum index in c_1' than they do in c_2' and assumes a chain of equalities $x_i = x_{i+1}$ that "catch up" to the max indices of c_2'. The residual r_2 is symmetric. We define merge formally below

$$\begin{aligned}
\mathsf{merge}(\mathcal{I}_1, \mathcal{I}_2) \quad \triangleq \quad & \textbf{let } r_1 \textbf{ be } \mathsf{asm}\,(\textstyle\bigwedge\{x_i = x_{i+1} \mid \mathcal{I}_1(x) \leq i < \mathcal{I}_2(x), x \in \mathsf{Var}\}) \textbf{ in}\\
& \textbf{let } r_2 \textbf{ be } \mathsf{asm}\,(\textstyle\bigwedge\{x_i = x_{i+1} \mid \mathcal{I}_2(x) \leq i < \mathcal{I}_1(x), x \in \mathsf{Var}\}) \textbf{ in}\\
& \textbf{let } \mathcal{J} \textbf{ be } \{x \mapsto \max\{I_1(x), I_2(x)\} \mid x \in \mathsf{Var}\} \textbf{ in}\\
& (r_1, r_2, \mathcal{J})
\end{aligned}$$

Note that the size of the added residuals is quadratic in the size of the input program [46]. Of course the translation is semantics-preserving, after some book-keeping to relate the lowest and highest indices with the inputs and outputs of the original program [46].

To understand passify by example, let's return to Figure 5.2, for which we compute the following:

GCL Model

```
$\langle a, g\rangle := \mathit{Group}(\mathsf{ipv4.dst});$
$\mathsf{grp} := g;$
$\langle b, p\rangle := \mathit{Agg}(\mathsf{grp});$
$\mathsf{if}(b = \mathsf{set\_port})\{$
  $\mathsf{port} := p$
$\}\{$// *else* $b$ *is* nop
  skip
$\}$
```

$$\xmapsto{\pi_1 \circ \mathsf{passify}(-, \mathcal{Z})}$$

Passive Form GCL Model

```
asm $\langle a_1, g_1\rangle = \mathit{Group}(\mathsf{ipv4.dst}_0)$
asm $\mathsf{grp}_1 = g_1;$
asm $\langle b_1, p_1\rangle = \mathit{Agg}(\mathsf{grp}_1)$
$\mathsf{if}(b_1 = \mathsf{set\_port})\{$
  asm $\mathsf{port}_1 = p_1$
$\}\{$// *else* $b_1$ *is* nop
  asm $\mathsf{port}_1 = \mathsf{port}_0$
$\}$
```

Notice the residual that was added to the nop branch of the choice operator. Because the set_port branch in the original program (above left) updated port to d, the passive equivalent incremented port's index to 1. Now, to synchronize the indices across branches, and capture that the value remained unchanged, passify adds the residual $\mathsf{asm}\,\mathsf{port}_1 = \mathsf{port}_0$ to the nop branch.

Assuming that a program is in passive form, we can generate a linear-size symbolic representation[2]. The following symbolic compilation function $N : \mathrm{GCL}(\mathcal{T}) \to \mathcal{T}$, precisely captures the executions of a passive program c:

$$\begin{aligned} N(\mathsf{asm}\,\varphi) &\triangleq \varphi \\ N(p_1; p_2) &\triangleq N(p_1) \wedge N(p_2) \\ N(p_1 \,\square\, p_2) &\triangleq N(p_1) \vee N(p_2) \end{aligned}$$

The following shows the result of running N on the passified example program:

Passive Form GCL Model

```
asm ⟨$a_1, g_1$⟩ = $Group(\mathsf{ipv4.dst}_0)$
asm $\mathsf{grp}_1 = g_1$;
asm ⟨$b_1, p_1$⟩ = $Agg(\mathsf{grp}_1)$
if($b_1$ = set_port){
  asm $\mathsf{port}_1 = p_1$
}{// else $b_1$ is nop
  asm $\mathsf{port}_1 = \mathsf{port}_0$
}
```

$\overset{N}{\longmapsto}$

Symbolic Model

$$\begin{aligned} &\langle a_1, g_1\rangle = Group(\mathsf{ipv4.dst}_0) \wedge \\ &\mathsf{grp}_1 = g_1 \wedge \\ &\langle b_1, p_1\rangle = Agg(\mathsf{grp}_1) \wedge \\ &(b_1 = \mathsf{set\_port} \wedge \mathsf{port}_1 = p_1 \\ &\quad \vee\ b_1 = \mathsf{nop} \wedge \mathsf{port}_1 = \mathsf{port}_0) \end{aligned}$$

With a symbolic pipeline in hand, we can check whether it satisfies a spec φ via implication. However, we must be sure to update φ with respect to passify's

[2]The standard presentation of compact symbolic compilation [46] also uses an additional *wrong execution function* W which captures when programs violate assert statements. But GCL($\mathcal{T}$) has no assertions, so it can never "go wrong."

output index mapping $\mathcal{I}$, that is $\mathsf{subst}(\varphi, \mathcal{I})$. In our example, since $\mathcal{I}(\mathsf{port}) = 1$, we check the following:

$$\begin{pmatrix} \langle a_1, g_1 \rangle = \mathit{Group}(\mathsf{ipv4.dst}_0) \wedge \\ \mathsf{grp}_1 = g_1 \wedge \\ \langle b_1, p_1 \rangle = \mathit{Agg}(\mathsf{grp}_1) \wedge \\ (b_1 = \mathsf{set\_port} \wedge \mathsf{port}_1 = p_1 \\ \quad \vee \; b_1 = \mathsf{nop} \wedge \mathsf{port}_1 = \mathsf{port}_0) \end{pmatrix} \Rightarrow \quad \mathsf{port}_1 \neq \mathsf{NONE}$$

We define symbolic compilation using $\text{VCG}\textsc{en} : \text{GPL}(\mathcal{T}) \times \mathcal{T} \to \mathcal{T}$, as shown below:

$$\begin{aligned} \text{VCG}\textsc{en}(p, \varphi) \triangleq \; & \textbf{let } c \textbf{ be } \mathsf{model}(p) \textbf{ in} \\ & \textbf{let } c', \mathcal{I} \textbf{ be } \mathsf{passify}(c, \mathcal{Z}) \textbf{ in} \\ & N(c') \Rightarrow \mathsf{subst}(\varphi, \mathcal{I}) \end{aligned}$$

We prove that VCGen is a precise ci-spec:

Theorem 5.2.2 (Symbolic Compilation)**.** $\text{VCG}\textsc{en}(p, \varphi)$ *is a precise ci-spec for* p *and* φ*.*

Proof. By Theorem 5.2.1 and [46]. □

Hence, the sentence $\text{VCG}\textsc{en}(p, \varphi)$ is a valid formula for validating configs. However, this formula, being almost a line-for-line translation of the initial problem $p \models \varphi$, with the added complexity of indexed variables, is not a significant improvement on the original program. Further, finding counterexamples for a concrete σ (i.e., satisfying $N(c) \wedge \neg \mathsf{subst}(\varphi, \mathcal{I})$) is NExpTime-complete [70]. Checking such a formula on every control-plane update could incur significant latency.

5.3 Computing Efficiently Control-Monitorable Sentences

Rather than repeatedly running NExpTime-complete checks, we propose a class (ECMS) of first-order sentences that can be checked efficiently—i.e., with polynomial complexity for a fixed set of typed functions $\mathbf{F}$. Specifically, we will characterize the complexity of monitoring an ECMS in terms of its *expression complexity*, a concept from database theory [1]. We then define an algorithm that computes a precise ci-spec, by leveraging quantifier elimination (Qe), making sure to show that this precise ci-spec is an ECMS. Finally, we show that any algorithm that computes a ci-spec in ECMS can solve the Qe for UFBV. This equivalence means that computing an ECMS may still incur a combinatorial blowup—i.e., the formula we generate will have exponential size in cases where bit-blasting is required. However, as shown in our experiments, we avoid bit-blasting in the common case. So working with formulae in ECMS is useful in practice.

First, we define a syntactic set of sentences that are efficiently monitorable by the control plane:

Definition 5.3.1 (Efficiently Control-Monitorable)**.** A sentence ψ of UFBV over a fixed set of functions $\mathbf{F} = \{F_1, \ldots, F_n\}$ is said to be *efficiently control-monitorable* ($\psi \in$ ECMS) if, for variable sets $\mathbf{z} = \{z_1, \ldots, z_n\}$, $\mathbf{y} = \{y_1, \ldots, y_n\}$ and $\mathbf{x} \subseteq \mathbf{z} \cup \mathbf{y}$, ψ can be written $z_1 = F_1(y_1) \wedge \cdots \wedge z_n = F_n(y_n) \Rightarrow \varphi(\mathbf{x})$ where $\varphi \in$ QFBV. For brevity, we write ψ as $\mathbf{z} = \mathbf{F}(\mathbf{y}) \Rightarrow \varphi(\mathbf{x})$.

To calculate the *expression complexity*,[3] one fixes the database, and expresses complexity in terms of the size of the query. In contrast, to calculate the *data complexity*, one fixes the query, and expresses complexity in terms of the size of

[3]Also called query complexity

the database. The *combined complexity* expresses complexity in terms of the sizes of both the query and the database [1]. In our setting, we focus on expression complexity. First, we fix the control plane interface to be a set $\mathbf{F} = \{F_1, \ldots, F_n\}$ and their associated types, e.g. $F_i : 2^{w_i} \to 2^{\ell_i}$. With a fixed $\mathbf{F}$, the number of functions n, and every w_i and ℓ_i are also fixed, which means a config σ comprises finite functions between fixed-size domains. We show the expression complexity is polynomial.

Theorem 5.3.1. *For a fixed config σ, and $\psi \in$ ECMS, checking $\sigma \models \psi$ is polynomial.*

Proof. Let $\psi \in$ ECMS. This means there is $\varphi \in$ QFBV, and variable sets $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ such that $\psi = \mathbf{z} = \mathbf{F}(\mathbf{y}) \Rightarrow \varphi(\mathbf{x})$ and $\mathbf{x} \subseteq \mathbf{y} \cup \mathbf{z}$. First, observe that given a valuation $\mu : \mathbf{x} \to 2^*$, checking $\models_\mu \varphi(\mathbf{x})$ is polynomial in the size of φ—simply evaluate the formula. Since σ is fixed, $M = \sigma(F_1) \times \cdots \times \sigma(F_n)$ has a fixed size. Each element $\mu_i \in M$ corresponds to a valuation $\mu_i : \mathbf{z} \cup \mathbf{y} \to 2^*$, and since $\mathbf{x} \subseteq \mathbf{z} \cup \mathbf{y}$, we can write $\mu_i : \mathbf{x} \to 2^*$. It suffices to perform the fixed number of polynomial checks $\models_{\mu_i} \varphi(\mathbf{x})$ for $i = 1, \ldots, |M|$. $\square$

The analysis above also shows that the data complexity, and hence the combined complexity, is exponential in the size of σ. However, this only captures the uninteresting observation that in the worst case, a monitor must inspect every combination of elements in a pipeline's tables. The expression complexity captures the complexity of each validation.

We show, in the remainder of this section, that rather than computing general first-order-logic formulae, it suffices to compute formulae in ECMS. We do this by showing that inferring a precise ECMS is formally equivalent to quantifier elimination (QE). While quantifier elimination algorithms normally define QE using an existential quantifier variable and use structural recursion to define it over the full grammar, it's more convenient to use the universal variant of QE as below:

Definition 5.3.2 (Quantifier Elimination)**.** Given a formula $\varphi(x_0, \mathbf{x}) \in$ QFBV, with $x_0 \in \mathsf{Var}$, $\mathbf{x} \subseteq \mathsf{Var}$ and $x_0 \notin \mathbf{x}$, a solution to the quantifier elimination problem is a formula $\psi(\mathbf{x})$ on only the variables $\mathbf{x}$ such that $\psi(\mathbf{x}) \Leftrightarrow \forall x_0 . \varphi(x_0, \mathbf{x})$. We write $\psi(\mathbf{x}) = \textsc{Qe}(\forall x_0 . \varphi(x_0, \mathbf{x}))$.

A corollary of our construction in the following sections will be that restricting ci-specs to ECMS does not affect the expressiveness. That is, computing the weakest ECMS is equivalent to computing the weakest first-order *sentence*.

5.3.1 QE Computes Precise ci-specs

To infer a precise ECMS constraining configs σ such that $p[\sigma] \models \varphi$, we compile p to a GCL program c, and then lift out the functions. We define a lift function that, loosely speaking, separates out the control plane (i.e. the tables) from the data plane (the forwarding behavior). To do this, we introduce ghost variables $\mathbf{z}$ and $\mathbf{y}$ that capture the inputs and outputs of the tables $\mathbf{t}$. Then we write $\mathbf{z} = \mathbf{t}(\mathbf{y})$ to nondeterministically capture all potential table rows—this space is collapsed to the runtime key x in the data plane program. We use $\mathbf{x}$ to indicate the remaining variables that occur in c. Formally we write $\mathsf{lift}(c) = \langle \mathbf{z} = \mathbf{t}(\mathbf{y}), d \rangle$ to indicate the following, lifted in the expected way:

$$\mathsf{lift}\left(\langle a, d\rangle = t(x); \mathsf{run}_{\mathbf{a}}(a, d)\right) \triangleq \langle \langle z_a, z_d \rangle = t(y_x), \mathsf{asm}\, y_x = x; \mathsf{run}_{\mathbf{a}}(y_a, y_d) \rangle$$

Notice that the output program d has no uninterpreted functions, that is $c' \in$ GCL(QFBV). The relationship between d and c can be captured below:

$$(\mathsf{asm}\,(\mathbf{z} = \mathbf{t}(\mathbf{y}))\,; d) \equiv_{\mathbf{x}} c \tag{5.1}$$

where $\equiv_{\mathbf{x}} \subseteq$ GCL $\times$ GCL relates programs that are equivalent on the variables $\mathbf{x}$. We can see this relationship by running lift on our example from Figure 5.2 as below:

$$\begin{array}{cl} & \mathsf{asm}\, y_1 = \mathsf{ipv4.dst}; \\ \langle z_a, z_g\rangle = \mathit{Group}(y_1) & \\ & \mathsf{grp} := z_g; \\ \wedge & \\ & \mathsf{asm}\, y_2 = \mathsf{grp}; \\ \langle z_b, z_p\rangle = \mathit{Agg}(y_2) & \\ & \mathsf{if}(z_b = \mathsf{set\_port})\{\mathsf{port} := z_p\}\{\mathsf{skip}\} \end{array}$$

The formula on the left "queries" the pipeline's interface with the *Group* and *Agg* tables, using the ghost variables $\mathbf{y}$ and $\mathbf{z}$ to capture the results. Then, the program on the right uses these variables to capture the forwarding behavior. Below, we recombine these components according to Equation (5.1):

$$\begin{array}{lcl} & & \mathsf{asm}\left(\begin{array}{c}\langle z_a, z_g\rangle = \mathit{Group}(y_1)\wedge \\ \langle z_b, z_p\rangle = \mathit{Agg}(y_2)\end{array}\right); \\ \langle a, g\rangle := \mathit{Group}(\mathsf{ipv4.dst}); & & \\ \mathsf{grp} := g; & & \mathsf{asm}\, y_1 = \mathsf{ipv4.dst} \\ \langle b, p\rangle := \mathit{Agg}(\mathsf{grp}); & & \mathsf{grp} := z_g; \\ \mathsf{if}(b = \mathsf{set\_port})\{ & \equiv_{\mathsf{ipv4.dst,grp,port}} & \mathsf{asm}\, y_2 = \mathsf{grp}; \\ \quad \mathsf{port} := p; & & \mathsf{if}(z_b = \mathsf{set\_port})\{ \\ \}\{\mathsf{skip}\} & & \quad \mathsf{port} := z_p \\ & & \}\{\mathsf{skip}\} \end{array}$$

Notice that in the lifted program on the right we've lifted all function calls to the start of the program. We then use the assumptions like $\mathsf{asm}\, y_1 = \mathsf{ipv4.dst}$ to collapse the space of lookups to precisely those where we looked up the value of $\mathsf{ipv4.dst}$ in the function *Group*.

Because the lifting stage preserves equivalence on the relevant variables (Equation (5.1)), it will intercede after the modeling stage. To evoke the fact that lift separates the control plane from the data plane, we will define a control plane symbolic compilation function $\mathcal{C} : \text{GPL} \to \text{UFBV}$ and a data plane symbolic compilation function $\mathcal{D} : \text{GPL} \times \text{QFBV} \to \text{QFBV}$. We define these below:

$$\begin{aligned}
\mathcal{C}(p) \triangleq\ & \textbf{let } c \textbf{ be } \mathsf{model}(p) \textbf{ in} \\
& \textbf{let } \vartheta, d \textbf{ be } \mathsf{lift}(c) \textbf{ in} \\
& \vartheta
\end{aligned}
\qquad
\begin{aligned}
\mathcal{D}(p, \varphi) \triangleq\ & \textbf{let } c \textbf{ be } \mathsf{model}(p) \textbf{ in} \\
& \textbf{let } \vartheta, d \textbf{ be } \mathsf{lift}(c) \textbf{ in} \\
& \textbf{let } d', \mathcal{I} \textbf{ be } \mathsf{passify}(d, \mathcal{Z}) \textbf{ in} \\
& \textbf{let } \varphi' \textbf{ be } \mathsf{subst}(\varphi, \mathcal{I}) \textbf{ in} \\
& N(d') \Rightarrow \varphi'
\end{aligned}$$

Both of these functions start the same, by modeling $p \in \text{GPL}$ as a GCL program c and then lifting the control plane ϑ out of the data plane d. The control plane function $\mathcal{C}$, stops here and returns ϑ. The data plane function continues its symbolic compilation, by computing a passive version d' of d by calling $\mathsf{passify}(d, \mathcal{Z})$ (recall that $\mathcal{Z}$ zero-initializes all passivization indices). Then, the data plane function normalizes the spec φ corresponding to the output indices $\mathcal{I}$, which produces φ'. Finally, $\mathcal{D}$ returns the formula $N(d') \Rightarrow \varphi'$.

The following lemma shows that $\mathcal{C}$ and $\mathcal{D}$ precisely characterize pipelines:

Lemma 5.3.2 (Lifting)**.** $\text{VCG}\textsc{en}(p, \varphi) \iff \mathcal{C}(p) \Rightarrow \mathcal{D}(p, \varphi)$

Proof. By Equation (5.1) □

The final step is to use $\textsc{Qe}$ to eliminate the packet variables $\mathbf{x}$ from $\mathcal{D}(p, \varphi)$. Since $\mathcal{D}(p, \varphi)$ is a formula over the original data plane variables $\mathbf{x}$ as well as on the ghost variables $\mathbf{y}$ and $\mathbf{z}$, the result of using $\textsc{Qe}$ to eliminate $\mathbf{x}$, will be a formula $\psi(\mathbf{y}, \mathbf{z})$ over just the variables $\mathbf{y}$ and $\mathbf{z}$. In fact, a key result in the domain of logical abduction [35, 33] is that $\psi(\mathbf{y}, \mathbf{z})$ is the *weakest* formula on the variables $\mathbf{y}$ and $\mathbf{z}$ such that $\psi(\mathbf{y}, \mathbf{z}) \Rightarrow \mathcal{D}(p, \varphi)$. Combining this weakness with the fact that $\textsc{Qe}$ is equivalence-preserving, we can see that $\textsc{Qe}$ suffices to solve the ci-spec inference problem.

Formally, we define a procedure $\textsc{PrecSpec}(p, \varphi)$ as follows:

$$\textsc{PrecSpec}(p, \varphi) \triangleq \mathcal{C}(p) \Rightarrow \textsc{Qe}(\forall \mathbf{x}. \mathcal{D}(p, \varphi))$$

where $\mathbf{x} = \mathsf{Var} \setminus \mathbf{y}$ where $\mathbf{y}$ is the set of all ghost variables that occur in $\mathcal{C}(p)$. Said another way, $\mathbf{x}$ is the set of indexed data plane variables.

Now, based on the observations we've made so far, we can prove that $\textsc{PrecSpec}(p, \varphi)$ precisely captures the control plane configs σ that make p satisfy its spec φ:

Theorem 5.3.3. $\textsc{PrecSpec}(p, \varphi)$ *is a precise ci-spec.*

Proof. By Lemma 5.3.2, [35, 33], and Definition 5.3.2. $\square$

Finally, by examining its syntax, we'll see that $\textsc{PrecSpec}(p, \varphi) \in$ ECMS! Here's how: since $\mathcal{C}(p)$ can be written as $\mathbf{z} = \mathbf{F}(\mathbf{y})$, and since $\varphi \in$ QFBV, then $\textsc{Qe}(\forall x. \mathcal{D}(p, \varphi))$ is QFBV. Further, since $\textsc{PrecSpec}(p, \varphi)$ is indeed precise, the fact that it is also in ECMS means that we have not given up any precision in restricting our ci-specs to be efficiently monitorable.

At first blush, it seemed that ci-spec inference would require us to learn arbitrary first-order logic formulae. We've shown here that it suffices to learn formulae in ECMS, and specifically, that we need only eliminate quantifiers in the theory of bitvectors.

5.3.2 Precise ci-spec Inference in ECMS Solves $\textsc{Qe}$

Unfortunately, we can show that precise ci-spec inference in ECMS solves $\textsc{Qe}$ in the theory of bitvectors—whose best known algorithms [31, 7] require bit-blasting the finite domain of quantification. Hence, for ci-spec inference, we also resort to bit-blasting in the worst case.

Consider a formula $\varphi \in$ QFBV. We want to compute $\textsc{Qe}(\forall x_0.\, \varphi(x_0, \mathbf{x}))$. To do this, we will use the GPL program $t(\mathbf{x})$ where $t : 2^{|x_1|+\cdots+|x_n|} \to \{\lambda().\mathsf{skip}\}$, and compute its ci-spec w.r.t. $\varphi(x_0, \mathbf{x})$. Now, $\textsc{PrecSpec}(t(\mathbf{x}), \varphi(x_0, \mathbf{x}))$ gives us the following formula:[4]

$$\langle z_a, z_d \rangle = t(\mathbf{y}) \Rightarrow \textsc{Qe}(\forall x_0, \mathbf{x}.\, \mathbf{y} = \mathbf{x} \Rightarrow \varphi(x_0, \mathbf{x})) \tag{5.2}$$

Notice that the call to $\mathsf{run}_{\mathbf{a}}(z_a, z_d)$ has disappeared. This is because the choice to run one of the actions in the singleton set $\{\mathsf{skip}\}$ will deterministically run that single action. As a result, z_a and z_d do not occur except for in the leftmost assumption. So, we can use the so-called "one-point rule" (also known as destructive equality resolution), to rewrite Equation (5.2) into the following:

$$\textsc{Qe}(\forall x_0, \mathbf{x}.\, \mathbf{y} = \mathbf{x} \Rightarrow \varphi(x_0, \mathbf{x})) \tag{5.3}$$

Next, we apply the one-point rule again and swap $\mathbf{y}$ for $\mathbf{x}$, which then lets us eliminate the innermost $\forall \mathbf{x}$, since the variables in $\mathbf{x}$ no longer occur. We that the following formula:

$$\textsc{Qe}(\forall x_0 . \varphi(x_0, \mathbf{x})) \tag{5.4}$$

which is equivalent to our original sentence. Having just proved it, we state the theorem below.

[4]Technically, $\textsc{PrecSpec}$ computes a formula where each variable has a passive index of 0, that is x_{00}, $\mathbf{x}_0$, $\mathbf{y}_0$, but by erasing the indices, we get the formula shown in Equation (5.2)

Theorem 5.3.4. $\textsc{PrecSpec}(t(\mathbf{y}), \varphi(\mathbf{x}, \mathbf{y})) = \forall \mathbf{y}.\textsc{Qe}(\forall \mathbf{x}.\varphi(\mathbf{x}, \mathbf{y}))$ *where* $t : 2^{|y_1|+\cdots+|y_n|} \to \{\lambda().\mathsf{skip}\}$.

Proof. As above. □

The downside of having shown the equivalence of Qe and ci-spec inference in ECMS is that the best-known algorithms resort to bit-blasting in the worst case. However, in what follows, we exploit domain insights to develop an algorithm that can eliminate quantifiers effectively.

5.4 Programmatic Qe

Since inferring ci-specs is intractable in general, we pursue heuristic techniques that work well in practice. A standard maneuver when dealing with large, intractable problems is to decompose the problem into smaller, easier-to-solve, subproblems. We exploit the fact that ci-spec inference commutes with choice (i.e., $\square$). That is, given a GPL program $p_1 \square p_2$ and a spec φ, it is the case that $\textsc{PrecSpec}(p_1 \square p_2, \varphi) \Leftrightarrow \textsc{PrecSpec}(p_1, \varphi) \wedge \textsc{PrecSpec}(p_2, \varphi)$. By reasoning inductively, this relationship can be generalized over all paths: $\textsc{PrecSpec}(p, \varphi) \Leftrightarrow \mathcal{C}(p) \Rightarrow \bigwedge_{\pi \in \mathsf{paths}(p)} \textsc{PrecSpec}(\pi, \varphi)$. We define $\mathsf{paths} : \text{GCL} \to \mathcal{P}(\text{GCL})$ below:

$$\begin{aligned}
\mathsf{paths} &: \text{GCL} \to \mathcal{P}(\text{GCL}) \\
\mathsf{paths}(x := e) &\triangleq \{x := e\} \\
\mathsf{paths}(\mathsf{asm}\,\varphi) &\triangleq \{\mathsf{asm}\,\varphi\} \\
\mathsf{paths}(c_1; c_2) &\triangleq \{\pi_1; \pi_2 \mid \pi_i \in \mathsf{paths}(c_i), i = 1, 2\} \\
\mathsf{paths}(c_1 \square c_2) &\triangleq \mathsf{paths}(c_1) \cup \mathsf{paths}(c_2)
\end{aligned}$$

Notice that we have defined paths on the GCL level. That is for a program $c \in \text{GCL}$, $\mathsf{paths}(c) \subseteq \text{GCL}(\mathcal{T})$ is the set of straight-line programs (aka paths) through c. We then define $\mathsf{paths}(p)$ for a program $p \in \text{GPL}$ by first compiling p to its data plane-only representation using model and lift. That is $\mathsf{paths}(p) = \mathsf{paths} \circ \pi_2 \circ \mathsf{lift} \circ \mathsf{model}(p)$. Further, we have defined $\pi \in \text{GCL(UFBV)}$. We define $\textsc{PrecSpec}(\pi, \varphi)$ to be $\textsc{Qe}(\forall \mathbf{x}. \mathcal{D}(\pi, \varphi))$, where $\mathbf{x}$ is the set of non-ghost variables in $\mathcal{D}(\pi, \varphi)$.

5.4.1 Paths Produce Smaller QE Problems

Computing the ci-spec for a single path is much more tractable than doing so for a whole program. Aside from being much smaller programs, aggressive compiler optimizations are much more powerful on paths. For instance, we use standard compiler transformations for dead code elimination and expression propagation. The dead code elimination function $\mathsf{dce} : \text{GCL} \times \mathsf{Var} \to \text{GCL} \times \mathsf{Var}$, at every step, removes assignments $x := e$ when x is not in the set of read variables R. It is defined below:

$$\begin{aligned}
\mathsf{dce}(x := e(\mathbf{y}), R) &\triangleq \begin{cases} \langle \mathsf{asm\,true}, R \rangle & x \notin R \\ \langle x := e, \mathbf{y} \cup (R \setminus \{x\}) \rangle & x \in R \end{cases} \\
\mathsf{dce}(\mathsf{asm}\,\varphi(\mathbf{y}), R) &\triangleq \langle \mathsf{asm}\,\varphi(\mathbf{y}), \mathbf{y} \cup R \rangle \\
\mathsf{dce}(c_1; c_2, R) &\triangleq \textbf{let } c_2', R_2 \textbf{ be } \mathsf{dce}(c_2, R) \textbf{ in} \\
& \textbf{let } c_1', R_1 \textbf{ be } \mathsf{dce}(c_1, R_2) \textbf{ in} \\
& \langle c_1'; c_2, R_1 \rangle
\end{aligned}$$

Similarly, $\mathsf{prop} : \text{GCL} \times \text{GCL} \to \text{GCL}$ propagates substitutions $x := e$ by substituting e for x in the rest of the path. This substitution must be done carefully to avoid "capture". For an imperative path like this one, substitution stops once x

appears on the left-hand side of an assignment. This definition differs from typical definitions of constant or expression propagation, which need to merge sets of facts at join points. Because we're reasoning about straight-line code, the set of facts never diverges. We define it formally below:

$$\begin{aligned}
\mathsf{prop}(x := e, c) &\triangleq c[x \mapsto e] \\
\mathsf{prop}(\mathsf{asm}\,\varphi, c) &\triangleq \mathsf{asm}\,\varphi; c \\
\mathsf{prop}(c_1; c_2, c_3) &\triangleq \mathsf{prop}(c_1, \mathsf{prop}(c_2, c_3))
\end{aligned}$$

These compiler optimizations are actually doing heuristic quantifier elimination *at the program level.* Notice that after the lifting stage, control plane variables will *never* occur in assignments, only the data plane variables will. So, using dce and prop to eliminate as many assignments as possible before running QE is a clear advantage of path decomposition.

However, the ability to generate smaller and more-optimizable QE instances doesn't mean much if there are exponentially many of them to solve. Since $\mathsf{paths}(p)$ is exponential[5] in the size of p, it remains intractable to compute $\textsc{PrecSpec}(\pi, \varphi)$ for every $\pi \in \mathsf{paths}(p)$.

Luckily, we don't always need to examine every program path. In fact, we only need to explore paths that violate φ. Since the ci-spec for a path that doesn't violate φ is $\top$, then $\textsc{PrecSpec}(c, \varphi)$ is equivalent to the ci-spec for only the paths that violate φ. Let's call this set of buggy paths B. In our experience (Section 5.7.1), we've seen that the number of these "buggy" paths can be orders of magnitude smaller than the size of $\mathsf{paths}(p)$.

Furthermore, we don't even need to analyze every buggy path in B. In fact, our experience has shown (Section 5.7.4) that the ci-spec for a single path generalizes

[5]The well-known *path explosion problem.*

to solve many paths. For instance, if we added a parser to our example from Figure 5.2 that either validated one of the main Layer 4 protocols, TCP or UDP, then the ci-spec for either parser path would generalize to the other.

5.4.2 A Path-Based Iterative Strengthening Algorithm

Our algorithm, CegQe, iteratively strengthens a candidate ci-spec using counterexamples. The procedure Strengthen takes in a spec φ, a candidate ci-spec ψ_i, and a spec-violating path $\pi \models \psi_i \wedge \neg\varphi$, and computes a new candidate ci-spec ψ_{i+1} such that $\psi_{i+1} \Rightarrow \psi_i$, and $\pi \models \psi_{i+1} \Rightarrow \varphi$. We define Strengthen as follows:

$$\textsc{Strengthen}_\pi(\psi, \varphi) \triangleq \psi \wedge \textsc{PrecSpec}(\pi, \varphi)$$

By definition, $\textsc{Strengthen}_\pi(\psi, \varphi) \Rightarrow \psi$. Similarly, since $\pi \models \textsc{PrecSpec}(p, \varphi) \Rightarrow \varphi$, then $\pi \models \textsc{Strengthen}_\pi(\psi, \varphi) \Rightarrow \varphi$, indicating that the strengthened ci-spec prohibits π from violating φ.

The algorithm iteratively strengthens ψ until an SMT solver proves ψ is stronger than $\mathcal{D}(p, \varphi)$. To maintain precision, $\textsc{Strengthen}_\pi$ will never "overshoot" $\mathcal{D}(p, \varphi)$. That is, as long as $\pi \in \mathsf{paths}(p)$, the invariant $\mathcal{D}(p, \varphi) \Rightarrow \textsc{Strengthen}_\pi(\psi, \varphi)$ holds. This formula is similar to `bf4`'s necessity constraint (which they write $OK \models \phi$, where ϕ is a new candidate ci-spec). While `bf4` checks this constraint using an SMT solver after each operation, Capisce maintains this as invariant, which holds because the conjunction of path-based ci-specs is equivalent to the full program ci-spec.

We define the following set $\textsc{BadPath}_p$ to capture all paths π that witness the insufficiency of ψ to prove $\mathcal{D}(p, \varphi)$. In practice, we use an SMT solver to produce one such path, when it exists.

$$\textsc{BadPaths}_p(\psi, \varphi) \triangleq \{\pi \in \mathsf{paths}(p) \mid \pi \models \psi \wedge \neg\mathcal{D}(p, \varphi)\}$$

Now, the algorithm can be stated formally. For a program p, a spec φ, and a candidate ψ, define:

```
$\textsc{CegQe}(p, \varphi) \triangleq$
  $\psi \leftarrow \top$;
  **while** $\pi \in \textsc{BadPaths}_p(\psi, \varphi)$ :
    $\psi \leftarrow \textsc{Strengthen}_\pi(\psi, \varphi)$;
  **return** $\psi$
```

For any program p and spec φ, the algorithm $\textsc{CegQe}(p, \varphi)$ terminates, because the set of paths through p is finite. By initializing ψ to be $\top$ we ensure that we will never overshoot the correct ci-spec. Similarly, the ψ produced by CegQe is the most precise ci-spec. Correctness of CegQe comes from the fact that the final path is equivalent to $\bigwedge_{p \in B} \textsc{PrecSpec}(\pi, \varphi)$ for some set of bad paths $B \subseteq \mathsf{paths}(p)$. We sketch the proof of this algorithm below:

Theorem 5.4.1 (Correctness)**.** $\textsc{PrecSpec}(p, \varphi) \iff \mathcal{C}(p) \Rightarrow \textsc{CegQe}_{p,\varphi}(\top)$

Proof Sketch. CegQe terminates because it explores a finite set of paths—the continued strengthening of the candidate solution ensures that it never explores the same path twice. The forwards direction follows from the fact that $\textsc{CegQe}(p, \varphi)$ can be written as $\bigwedge_{\pi \in B}(\textsc{PrecSpec}(\pi, \varphi))$, for some set of bad paths $B \subseteq \mathsf{paths}(p)$. Since PrecSpec commutes with choice, the implication holds. The reverse direction follows by the emptiness of BadPaths which implies $\textsc{CegQe}(p, \varphi) \Rightarrow \mathcal{D}(p, \varphi)$. □

Finally, we have $\mathcal{C}(p) \Rightarrow \textsc{CegQe}(p, \varphi) \in$ ECMS since $\textsc{CegQe}(p, \varphi) \in$ QFBV.

All told, we've been able to reduce the size of each expensive $\textsc{Qe}$ sub-problem, by decomposing the program into its component paths, and using aggressive compiler optimizations to eliminate variables at the program level.

5.5 Specifications for Data Planes

The standard specification mechanism in data plane verification is assume-assert style specification [72]. Indeed, architectures [116, 54] for the P4 programming language have built-in functions called `assume` and `assert`. Even though they have no semantic effect on the program, programmers use these constructs with verification tools [72, 2] that reason about intermediate states of the system. Since GPL already has assumptions, we need only add assertions.

Unfortunately, adding assertions to GPL incurs a quadratic cost in the size of the formula [46], even along single paths. Assertions, written $\mathsf{ast}\,\varphi$, characterize when programs "go wrong" by violating φ. The function $W : \mathrm{GCL}(\mathcal{T}) \to \mathcal{T}$, originally defined by Flanagan & Saxe [46], symbolically characterizes these executions for a passive program $c \in \mathrm{GCL}(\mathcal{T})$. It is defined below:

$$\begin{aligned}
W(\mathsf{ast}\,\varphi) &\triangleq \neg\varphi \\
W(\mathsf{asm}\,\varphi) &\triangleq \bot \\
W(c_1; c_2) &\triangleq W(c_1) \vee N(c_1) \wedge W(c_2) \\
W(c_1 \mathbin{\square} c_2) &\triangleq W(c_1) \wedge W(c_2)
\end{aligned}$$

The quadratic size comes from the sequence rule. A program $c_1; c_2$ can either go wrong because c_1 goes wrong ($W(c_1)$), or because c_1 goes right ($N(c_1)$), and then c_2 goes wrong ($W(c_2)$).

Consequently, checking whether a candidate ψ suffices for a passive program c, with asserts, means checking $\psi \Rightarrow \neg W(c)$, which asks if ψ is sufficient to prove that the program never goes wrong. Because the spec φ has become part of the program (e.g. via a terminal ast statement), we don't have an explicit spec φ to reason about.

In the case that there is some $\pi \models \neg\psi \wedge W(c)$, we know that π that contains *at least one* violated assertion. Now we generate the following quantifier elimination problem, which eliminates the data plane variables $\mathbf{x}$ from the wrong executions (W) of the path π:

$$\textsc{Qe}\,(\forall \mathbf{x}.\ \neg W(\pi))$$

Unfortunately, $W(\pi)$ is quadratic in size [46]. In Section 5.4.2 we were checking the linear-size $N(\pi) \Rightarrow \varphi$. Now, we much larger $\textsc{Qe}$ instances. To sidestep this growth, and maintain the compactness of the $\textsc{Qe}$ sub-problems, we decompose the problem even further.

Let's proceed by example. For a path $c = c_1; \mathsf{ast}\,\varphi_1; c_2; \mathsf{ast}\,\varphi_2$, where c_1 and c_2 are ast-free, we would generate the following $\textsc{Qe}$ problem:

$$\textsc{Qe}\,(\forall x.\ \ (N(c_1) \Rightarrow \varphi_1) \wedge (N(c_1) \wedge \varphi \wedge N(c_2) \Rightarrow \varphi_2))$$

Observing DeMorgan's laws and the distributivity of $\textsc{Qe}$ and $\forall$ over conjunction, this becomes:

$$\textsc{Qe}(\forall \mathbf{x}. N(c_1) \Rightarrow \varphi_1) \quad \wedge \quad \textsc{Qe}(\forall \mathbf{x}. N(c_1) \wedge \varphi_1 \wedge N(c_2) \Rightarrow \varphi_2)$$

Projecting this reasoning back up into the program, each of these subproblems corresponds the following conjunction of assert-free ci-spec inference problems:

$$\textsc{PrecSpec}(c_1, \varphi_1) \quad \wedge \quad \textsc{PrecSpec}(c_1; \mathsf{asm}\,\varphi_1; c_2, \varphi_2)$$

In general, exploiting this distributivity lets us consider assert-final path *prefixes*. Given a counterexample packet *pkt*, our path generation scheme produces the path prefix that terminates in the *first* assertion that is violated. At the end, we find that even in the presence of assertions we will only ever produce *linear*-size QE sub-problems (although there can be many such sub-problems).

5.6 Implementation

We have implemented a ci-spec inference library in OCaml called Capisce. Our library exposes a GPL AST, which makes heavy use of smart constructors. Our algorithm largely follows the structure outlined in the previous section. We discuss here the implementation of path selection and quantifier elimination. We also discuss how GPL can model real world pipeline programs.

Path Selection The BADPATHS function in CEGQE(p, φ) checks whether there exist any buggy paths. To compute this check in practice, we use both an SMT solver, and a tracing execution of the program. First, we use an SMT solver to check $\mathrm{SAT}(\psi \wedge \neg \mathcal{D}(p))$, which returns a valuation of the input variables—that is, a packet *pkt*. We then define a tracing execution that runs *pkt* through the program p, accumulating its execution trace π as it goes. We then use π to strengthen ψ.

Quantifier elimination To eliminate quantifiers, we rely on an ensemble of state-of-the-art solvers: Z3 [31] and Princess [7]. In our experience, it seems that Z3 is more efficient at bit-blasting, while Princess is better at eliminating formulae with arithmetic operations ($+$, $-$, etc). We find that combining the respective strengths of these two solvers is highly effective. Rather than racing the solvers, as is common, we rely on the fact that both solvers produce partial results when they time out. We can then pass these partially-eliminated formulae between the two solvers. We have found that Z3;Princess;Z3 generally suffices.

5.7 Experience

To assess the usefulness of Capisce and its ci-specs, we investigate the following five research questions:

RQ1: Can CegQe to infer real safety properties?

RQ2: Are most program paths correct?

RQ3: Do ci-specs for individual paths generalize over many paths?

RQ4: Can ci-specs help programmers find bugs?

RQ5: How does Capisce compare to `bf4`?

In Section 5.7.1 we use Capisce to infer ci-specs for a suite of P4 programs collected from previous work [36], answering **RQ1** in the affirmative. Most ci-specs are inferrable in a few seconds, with several taking hours. Two programs, `fabric` and `linearroad`, reached our timeout of 24 hours, without having terminated. However, with simple fixes to each program, we can infer their ci-specs. We discuss these examples in Section 5.7.5.

To answer **RQ2** and **RQ3**, which refer to the guiding assumptions about the prevalence of bugs, we measure the proportion of paths that are covered after each strengthening step of CegQe. Our analysis in Section 5.7.4 shows that for the programs we analyzed, 40-96% of paths were initially correct. We also can see that the ci-spec for individual paths can generalize very well—in some cases over thousands of other paths.

To answer **RQ4**, we qualitatively analyzed the ci-specs Capisce computed. In Section 5.7.2, we discuss the programs that had absurd ($\perp$) ci-specs, that is, there was some buggy packet for every possible config. We were able to analyze these programs and fix their errors. We also analyzed the nontrivial specifications (Section 5.7.3), which revealed several hitherto-unknown bugs in the source programs. Further, a local analysis of `fabric`, on which Capisce timed out, directed us to fix bugs in its access control logic (Section 5.7.5).

To answer **RQ5**, we compared Capisce with `bf4` on our suite of programs (Section 5.7.6), and found that while Capisce is often much slower than `bf4`, its computed ci-specs are much safer.

We ran our experiments on a 64-core machine, with Intel Xeon Silver 4216 CPUs @ 2.10GHz, running Ubuntu 22.04. Each experiment was single-threaded.

5.7.1 Capisce in Practice

To understand the effectiveness of Capisce on real-world programs, we inferred Capisce for the programs used as benchmarks in previous work [36]. These benchmark programs comprise both research and industrial programs that are publicly available on Github.

First, we ran Capisce's inference with respect to two well-studied [72, 36, 37] properties in data plane programming. The first, called *Header Validity*, asserts

Program	Program Paths	Result	Time (s)	#Exp	Size	%Explored
			ABSURD PROGRAMS			
`ts-switching`	21	$\bot$	0.160	2	1	0.095
`mc-nat`	39	$\bot$	0.089	1	1	0.026
			FIXES TO ABSURD PROGRAMS			
`ts-switching-fixed`	21	$\top$	0.030	0	1	0.0
`mc-nat-fixed`	39	$\top$	0.027	0	1	0.0
			TRIVIAL PROGRAMS			
`resubmit`	9	$\top$	0.028	0	1	0.0
`netpaxos-acceptor`	116	$\top$	0.030	0	1	0.0
`ecmp`	102	$\top$	0.030	0	1	0.0
`hula`	3629	$\top$	0.068	0	1	0.0
`ndp-router`	3843	$\top$	2.9	0	1	0.0
			NONTRIVIAL PROGRAMS			
`arp`	95	φ	5.0	0.016	349	0.17
`heavy-hitter-2`	267	φ	0.29	3	26	0.011
`heavy-hitter-1`	327	φ	0.60	7	90	0.021
`flowlet`	649	φ	1.8	9	127	0.014
`simple_nat`	66531	φ	5.2	54	1421	0.00081
`07-multiprotocol`	54459	φ	16	143	3138	0.0026
`netchain`	26726780	φ	2.9×10^3	264	11658	9.9×10^{-6}
`linearroad`	54477696		timeout			
`fabric`	133365047559893		timeout			
			SPEC SMELL PROGRAM FIXES			
`heavy-hitter-1-fixed`	327	φ	0.63	7	107	0.021
`linearroad-fixed`	54477696	φ	5.9×10^4	3236	179885	5.9×10^{-5}
`fabric-fixed`	133365047559893	φ	1.2×10^3	653	41140	4.9×10^{-12}

Table 5.1: Experience with using Capisce to check Header Validity on a broad range of P4 programs.

that every header h is valid every time $h.f$ is read. The second, called *Determined Forwarding*, every packet is assigned forwarding behavior. In V1Model P4, which we use for our examples, we can check determined forwarding by ensuring that the variable `std_metadata.egress_spec` is assigned a value. As previous work has shown [37, 72], satisfying these properties requires complicated invariants on the control plane configs that potentially involve multiple tables, which makes them excellent benchmarking properties. The results can be seen in Tables 5.1 and 5.2. The "Result" column categorizes the result of running CEGQE on the program:

Program	Program Paths	Result	Time (s)	#Exp	Size	%Explored
		Absurd Programs				
`ecmp`	102	$\bot$	0.320	4	1	3.9
`fabric`	133365047559893	$\bot$	7.3	5	1	3.7×10^{-12}
`netchain`	26726780	$\bot$	27	7	1	2.6×10^{-5}
		Trivial Programs				
`arp`	95	$\top$	0.027	0	1	0.0
`linearroad`	54477696	$\top$	0.054	0	1	0.0
`simple-nat`	5548	$\top$	0.034	0	1	0.0
		Nontrivial Programs				
`resubmit`	9	φ	0.016	2	17	22
`ts-switching`	21	φ	0.10	1	4	4.8
`mc-nat`	39	φ	0.27	3	21	7.7
`netpaxos-acceptor`	116	φ	0.12	1	4	0.86
`heavy-hitter-2`	267	φ	88	15	233	5.6
`heavy-hitter-1`	327	φ	0.10	11	187	3.4
`flowlet`	649	φ	79	15	490	2.3
`hula`	3629	φ	0.39	1	9	0.028
`ndp-router`	3843	φ	40	36	824	0.94
`07-multiprotocol`	54459	φ	30	232	5034	0.43
		Spec Smells & Fixes				
`ecmp-fixed`	102	φ	0.28	3	34	2.9
`mc-nat-fixed`	27	$\top$	0.029	0	1	0.0

Table 5.2: Experience with using Capisce to check Determined Forwarding on a broad range of P4 programs.

$\top$ indicates that CegQe returned true; $\bot$ means that CegQe returned false, and φ captures everything in between. The "Time" column presents the number of seconds to 2 significant figures. We also report the number of paths through the original program, as well as the number (#Exp) and percent (%Explored) of concrete paths that CegQe explored.

Observe, first, that most of our programs have non-trivial and non-absurd specifications. These are programs that would have been rejected by standard verifiers [38, 72, 108, 112]. Second, observe that most of these programs have reasonable solve times—a few seconds to a few minutes. Even for the larger programs that have hundreds of millions to quadrillions of paths, we are only exploring a minute fraction of those paths.

5.7.2 True Data-Plane Bugs

For programs that produced empty control plane properties ($\perp$), we inspected the programs to understand the errors. For *Header Validity*, many of these programs with true data plane bugs had made implicit assumptions about the packets that would successfully pass the parser, which is a well-documented pattern [37]. We describe how we incorporated these assumptions in the Section 5.7.2. Conversely, for programs with determined forwarding bugs, the fix is to specify that by default the packet should be dropped at the start of egress processing, which trivializes all ci-specs.

Header Validity

In this section, we discuss the true data-plane bugs that we found in the `mc-nat` program. The `mc-nat` program is an industrial R&D program. The parser for this program performs standard Ethernet and IPv4 parsing, which means that at the start of the pipeline, Ethernet is known to be valid, but IPv4 may or may not be. The error occurs in the first table `set_mcg` shown below. Its key is `ipv4.dstAddr`, and therefore to instrument it for *Header Validity*, we have asserted the validity of `ipv4` (top right), which is not guaranteed by the parser (below left).

```
// Table Definitions        // Buggy Pipeline
set_mcg : bit<32> -> {...}  assert(ipv4.isValid = 1); // error!
// Parser                   set_mcg(ipv4.dstAddr); ...
eth.isValid = 1;
if (eth.type == 0x0800){    // Fixed Pipeline
  ipv4.isValid := 1; ...    assume(ipv4.isValid = 1); // fix!
} else {                    assert(ipv4.isValid = 1);
  ipv4.isValid := 0; ...    set_mcg(ipv4.dstAddr); ...
}
...
```

After inspecting the program, we concluded that the engineers only intended for this program to run on IPv4 packets. We realize this apparent assumption by adding an assume statement (shown above on the right). With this assumption, the assertion follows immediately, and Capisce returns the ci-spec `true`. The results for the fixed program are reported in Table 5.1 under `mc-nat-fixed`. The bug in `ts-switching` has a similar character and similar fix.

Determined Forwarding

We also found true violations of *Determined Forwarding*. One such example can be found in the `ecmp` program. Improving on the previous example, the `ecmp` program does guard the accesses to the optionally-valid `ipv4` header with an if statement. The problem is that when the `ipv4` header is invalid, no code is run, which means that the forwarding behavior is not determined. The following presents an outline of the `ecmp` program.

```
// Table Definition
table ecmp_group : bit<32> -> { ... }
// Pipeline
determined := 0;
if (ipv4.isValid == 1 && ipv4.ttl > 8w0) {
  ecmp_group(ipv4.dst); // may or may not determine forwarding
  ...
} else {
  <does NOT determine forwarding>
}
assert (determined == 1); // violated when the else branch runs
```

This bug has several fixes. We could set all packets to be dropped at the start of the pipeline, or we could manually determine the forwarding behavior in the else branch. After applying the latter fix, to produce `ecmp-fixed`, Capisce computes a sensible ci-spec in 280ms.

5.7.3 Bugs Found by Inspecting ci-specs

For most programs, Capisce computes non-trivial and non-absurd ci-specs (indicated by φ in Tables 5.1 and 5.2). We manually analyzed these specs, which gave us new insights about the programs. Concretely, we were able to discover real bugs in the programs. Borrowing the idea of *code smells*, we identify some simple "spec smells" that we have used to find bugs.

The first spec smell, called *prohibited action*, occurs when the inferred ci-spec prohibits one of the tables actions from ever occurring. It would be unusual for a programmer to implement an action that must never be used. The most likely explanation is that the program has a bug. The second smell, called *obligatory wildcard*, occurs when the inferred ci-spec requires a table to always wildcard one of its keys. Again, it would be unusual to declare a table with a useless key.

We return to `mc-nat`, which exemplifies the *prohibited action* code smell when analyzed w.r.t. the *Determined Forwarding* property. We then analyze `heavy-hitter-1` which exemplifies the *obligatory wildcard* code smell.

Prohibited Action

Returning to the `mc-nat` program, we will `set_mcg`, more closely. It has the following three actions: `set_output_mcg`, `drop`, and `nop` shown below.

```
// Definitions
action set_output_mcg (mcast_group : 16) =
  meta.mcast_group := mcast_group
action drop () = std_meta.egress_spec := 511
action nop () = skip
table set_mcg : bit<32> -> { set_output_mcg, drop, nop }
// Pipeline, IPv4 may or may not be valid
set_mcg(ipv4.dst); ...
```

Of `set_mcg`'s actions, only one that sets the egress specification: `drop`. Further, `set_output_mcg` sets `meta.mcast_group`, which triggers an assignment to the `egress_spec` field later in the pipeline. Finally, `nop` does neither, and the `egress_spec` remains undefined. As a result, Capisce computes a spec that prohibits `set_output_mcg` from running the `nop` action. This is a *prohibited action* smell, and a true bug. To fix it bug, we can simply remove `nop` from the actions list. After doing this, Capisce computes the trivial ci-spec—that is, $\top$—in 29ms.

Obligatory Wildcard

In the `heavy-hitter-1` program, we find an example of the *obligatory wildcard* spec smell when analyzing it w.r.t. *Header Validity*. The ci-spec computed by Capisce forces the `ipv4.dst` address to be wild-carded. We can examine the pipeline below to see why:

```
// Table Definitions
table count_table : bit<32> -> { ... }
table ipv4_lpm : bit<32> -> { ... }
table forward : bit<32> -> { ... }
// Pipeline --- ipv4 may or may not be valid
// To fix, assume ipv4.isValid = 1
count_table(ipv4.dst); ipv4_lpm(ipv4.dst); forward(nhop_ipv4);
```

After running a parser that optionally parses the IPv4 header (similar to the one

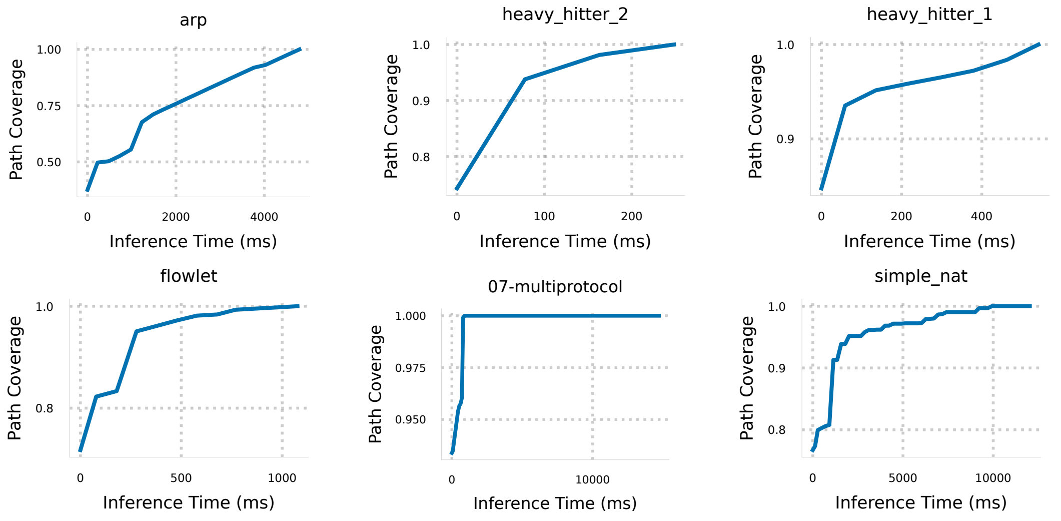


Figure 5.5: Path coverage over time for Header Validity analysis of programs with fewer than 100k paths.

shown in Section 5.7.2 for `mc-nat`), the `heavy-hitter-1` program immediately reads the `ipv4.dst` address. Since the `ipv4` header may be invalid, `count_table` *must not* read from it. Capisce recognizes this and forces `count_table` to wild-card its key. It seems strange that a single-key table should not be allowed to use any of its packet-classification power. This is likely not intended by the programmer, so we declare it a bug. We can fix it by assuming `ipv4`'s validity. After doing so, Capisce computes a sensible spec in 630ms.

5.7.4 Analyzing Path Decomposition

The majority of our programs had non-trivial ci-specs. Even for programs with quadrillions of paths, Capisce explores only a small fraction of them. In the extreme, for `fabric-fixed`, while we do explore nearly 41k paths, this is 12 orders of magnitude smaller than the number of paths through the program itself. While this fraction is extreme for our dataset, the rightmost columns of Tables 5.1 and 5.2 show that Capisce explores a small fraction of paths.

While Table 5.1 shows that it suffices to explore relatively few of a program's paths, we want a more fine-grained answer to our guiding assumptions (**RQ2** & **RQ3**). How many paths are actually buggy? How many paths are covered by each strengthening step?

To answer these questions, we measure how path coverage evolves as the candidate ci-spec gets stronger for a few of our small programs with nontrivial ci-specs. After the run finished, we measured the proportion of paths that satisfied the specification after assuming the new candidate spec—we call this proportion *path coverage*. We restricted ourselves to programs with fewer than 100k paths to make this analysis tractable.

The results of this analysis are shown in Figure 5.5. Notice the high proportion of safe paths when the inference time is 0. At the start the candidate ci-spec is $\top$, so the path coverage metric at this point is measuring the proportion of safe paths. The proportion of safe paths is very high, the lowest being 40% and the highest being nearly 95%. This empirical evidence supports our first guiding assumption: programs are usually correct. Second, notice the steep inclines early towards the left of the figures. These indicate that strengthening is highly effective—many other paths were covered by $\textsc{PrecSpec}(\pi, \varphi)$. For instance, in the `simple-nat` run, the ci-spec ψ_2 that was computed by the second bad path, π_2, covered approximately 10% of the remaining buggy paths. This empirical evidence supports our second guiding assumption: the ci-spec for a single path suffices to cover many other paths.

5.7.5 Limitations

So far, we've seen that despite the theoretical difficulty of ci-spec, Capisce computes useful ci-spec for real-world programs. Unfortunately, because ci-spec inference is theoretically difficult, it is unsurprising that Capisce hits the 24h timeout on 2 programs: `linearroad` and `fabric`. However, these timeouts can be considered their own "spec smells." In diagnosing why these programs reached the timeout, we found issues in the code. After fixing them, Capisce produced ci-specs.

Fabric

ONOS's `fabric.p4` is a production-grade L2/L3 data plane program. Originally used as a target for an internal API, it has evolved to be a mid-level abstraction layer [22], as well as support higher-level user plane functionality [73].

We were unable to compute a ci-spec for `fabric` in 24 hours (in fact, it took 16 days). In analyzing the ci-spec for subprograms, we detected the *obligatory wildcard* code smell in the `acl` table. Concretely, the ci-spec forces the `acl` table to always wild card `icmp.type` and `icmp.code`. This is because there is no way to for the controller to ensure that `hdr.icmp` is always valid.

The issue arises in `fabric`'s treatment of tunneling. After running the metadata initialization below on the left, `lkp` metadata holds the innermost valid IPv4 protocol and ICMP header data, as shown in the type and code below:

```
// Metadata Initialization
if (inner_ipv4.isValid = 1){
  lkp.ip_proto := inner_ipv4.proto;
  if (inner_icmp.isValid = 1){
    lkp.icmp_type := inner_icmp.type
        ;
    lkp.icmp_code := inner_icmp.code
        ;
  } else {}
} elif (ipv4.isValid = 1) {
  ...
  lkp.ip_proto := ipv4.proto;
  if (icmp.isValid = 1) {
    lkp.icmp_type := icmp.type;
    lkp.icmp_code := icmp.code;
  } else {...}
} else {...}
```

```
// Buggy Table Keys
acl(eth_type,
    lkp.ip_proto, ...,
    icmp.type, // buggy!
    icmp.code, // buggy!
    ... );
// Fixed Table Keys
acl(eth_type,
    lkp.ip_proto, ...,
    lkp.icmp_type, // fix!
    lkp.icmp_code, // fix!
     ...);
```

Now, in the `acl` table on the right, even though both `eth_type.value` and `lkp.ip_proto` appear in the keys, they are not sufficient to determine the validity of `icmp`. Together, these two keys can only determine that either `icmp` or `inner_icmp` is valid, but not which. Consequently, reads to the `icmp` header reads must be wild-carded. In the fixed version of the program, `fabric-fixed`, we replaced `icmp` and `icmp_code` with their respective `lkp` counterparts. With these fixes, Capisce computes its ci-spec in about 20 minutes.

Linearroad

Despite our best efforts to minimize QE problem instances, `linearroad`'s use of complex machine arithmetic causes our ensemble of QE solvers to resort to bit-blasting. This kind of computation is not typical in data plane programs. In fact, `linearroad` is an experimental program that was written to evaluate use of P4 for implementing streaming database queries [60].

The complex machine arithmetic arises in the update_ewma_spd table, which computes an estimated weighted moving average (EWMA). The relevant pieces of it are shown below:

```
seg_meta.ewma_spd :=
  seg_meta.ewma_spd * 96 + pos_report.spd * 16 >> 7
```

```
...
check_toll(..., seg_meta.ewma_spd, ...)
```

Since the value of the complicated machine arithmetic expression on the left flows into the key of the *CheckToll* table, we will need to reason about the possible values of seg_meta.ewma_spd. For instance, there are certain values, like 0xFE00, that will never be assigned to seg_meta.ewma_spd, this causes solvers to bit-blast to compute these values. However, in our inspection of the source code and the test cases, it's clear that the programmers did not intend for there to be any correctness requirements on this key. Capisce provides an annotation mechanism for keys that allows us to avoid bit-blasting and generate possibly less precise ci-specs. Programmers can annotate specific table keys as *unconstrainable*, which means that the ci-spec cannot reject configs based on the value of these keys. After marking seg_meta.ewma_spd unconstrainable, Capisce produces a ci-spec in under 17 hours, after exploring nearly 180k paths.

5.7.6 Comparison to `bf4`

We compared Capisce to the most relevant tool in previous work, `bf4`. To do this, we serialized the programs in our benchmarks as P4 programs and passed them into `bf4`.

To showcase Capisce's improvement over `bf4`, we analyzed *Header Validity* and *Determined Forwarding* over the benchmark suite programs that both tools agreed had bugs.[6] We used `bf4` to report the number of "bugs" (a.k.a. violable assertion points) that were reachable both before assuming the inferred ci-spec.

[6]For instance, `fabric` is omitted because `bf4` incorrectly marks it bug-free. Similarly, `hula` is excluded because `bf4` incorrectly detects bugs. We manually verified these analyses by inspecting the P4 code.

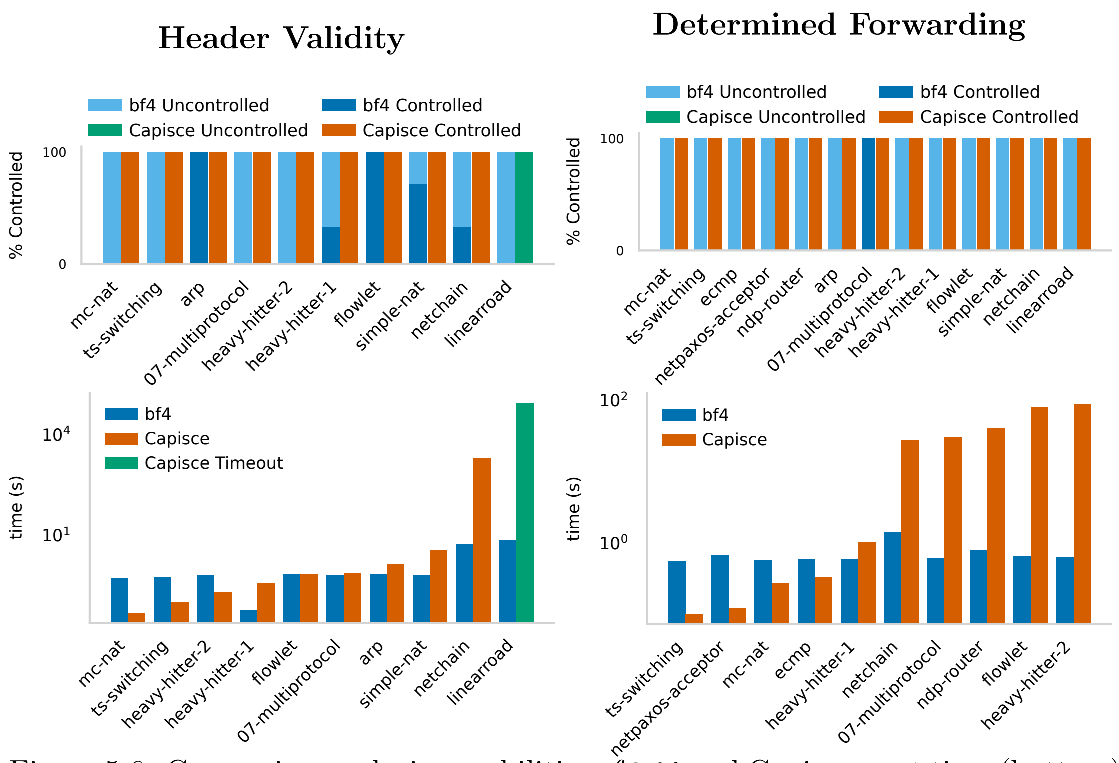


Figure 5.6: Comparing analysis capabilities of `bf4` and Capisce w.r.t time (bottom) and bugs controlled (top). Note the logarithmic y-axes on the time charts.

Then, each tool computed its ci-spec and reported the number of reachable bugs that remained. If a bug was not reachable after inferring the ci-spec, we say it was *controlled*. The results can be seen in Figure 5.6.

First, observe that for larger programs, `bf4` is orders of magnitude faster than Capisce. Note that the bar graphs at the bottom of Figure 5.6 have logarithmic y-axes. However, `bf4` can only control a fraction of the bugs that Capisce can. For *Header Validity*, while Capisce can control 96% of bugs, `bf4` can only control 40% of bugs. The only bugs that Capisce cannot control are the bugs for which it times out. For *Determined Forwarding*, Capisce controls 100% of bugs, while `bf4` controls only 1 out of 13 bugs for the programs in our benchmark suite.

In comparing Capisce with `bf4`, we have seen that with its extended run-times, Capisce can control many more bugs than can `bf4`. This is unsurprising, as tools have different goals: `bf4` quickly computes *necessary* ci-specs that maximize the number of controlled bugs, while Capisce produces safe (and indeed precise) ci-specs—that is, Capisce's ci-specs control all bugs.

Part III

Verified Configuration

CHAPTER 6

AUTOMATICALLY CONFIGURING THE DATA PLANE

The network control plane plays a similar role in modern systems as a classical OS kernel. It manages resources such as end-to-end forwarding paths, maps incoming traffic onto those paths, and enforces policy such as ensuring isolation between tenants in a public cloud.

One challenge that complicates the design of the control plane is dealing with data plane heterogeneity. Much as an OS kernel manages hardware resources for a variety of peripherals, the network control plane manages hardware resources for a variety of data planes. Most network operators purchase equipment from multiple manufacturers to avoid lock-in, which results in devices with heterogeneous feature sets, and even devices manufactured by the same vendor tend to evolve over time. This heterogeneity manifests as complexity throughout the control plane, appearing in low-level drivers and SDKs, device OSes (e.g., SONiC [106], FBOSS [26], Stratum [109]), higher-level APIs (e.g., OpenFlow [76], OpenConfig [88], P4Runtime [27]), and even network applications.

As an example, switches based on Broadcom ASICs such as Trident2, Tomahawk and Qumran-MX all expose an OpenFlow-like API to SDN controllers (or more precisely, the OF-DPA [90] abstraction). However, due to differences in the chips, the API behaves in subtly different ways on various devices. For instance, the Termination MAC table, which determines whether to route packets or bridge them, appears in all three devices but behaves differently on Trident2/Tomahawk versus Qumran-MX—the former supports matching on the ingress port while the latter does not. This discrepancy has led to bugs: before a special case was added to the ONOS controller, multicast traffic on Qumran-MX devices was flooded out on all ports rather than being forwarded to the proper multicast groups [94].

This anecdote is just one example of a more pervasive problem. The OF-DPA API specification [90] is more than 150 pages of English prose. The ONOS development team took two years to validate Qumran-MX switches and certify them as production-ready. This effort included multiple iterations of testing and bug fixing to port the Tomahawk driver to Qumran-MX, even though the devices come from the same vendor, implement the same protocols, and expose the same control plane abstractions. In practice, the problem of mapping abstract specifications of forwarding behavior down to real-world targets seems too hard to solve by hand.

Control Plane Synthesis. This paper presents a different approach to managing data plane diversity. Rather than relying on careful engineers to manually craft bug-free mappings from high-level abstractions to low-level targets, we show how to automate this task using program synthesis. More precisely, we develop Avenir, a system that automatically translates control plane operations written against an abstract forwarding specification (e.g., OF-DPA), into lower-level operations for a physical target (e.g., Qumran-MX).

Our approach proceeds in two steps. First, we use the P4 language [20] to model the behavior of the abstract and target devices. Although P4 was originally designed as a domain-specific language for programming devices like Barefoot's Tofino switch, it is also being used as a specification language for fixed-function devices (e.g., at Google [118]). For our purposes, what matters is that P4 provides a precise, bit-level specification of data plane behavior that can be mechanized using an SMT solver [72]. Hence, when P4 is not sufficiently expressive to model the pipelines' behavior, our approach should still be applicable. For example, one could work with other packet-processing languages like NPL, eBPF, or vendor

SDKs. Second, we use *counterexample guided inductive synthesis* (CEGIS) [105] to translate the abstract control plane operations, such as inserting entries into a match-action table, into equivalent physical operations. Our synthesis algorithm is provably *sound* (i.e., if it succeeds, the abstract and target behaviors are guaranteed to be equivalent) and *complete* (i.e., if there a translation for a given operation, Avenir finds it).

At a technical level, we exploit the insight that data plane devices are fundamentally simple. When modeled as programs, they lack complex features like pointers and loops (parser state machines and uses of recirculation can be finitely unrolled in practice). Although data planes exhibit complexity in other dimensions, such as the number of protocols or table entries they support, the amount of processing they perform on any given packet is limited. Hence, it is possible to model their behavior using simple, loop-free programs that are amenable to analysis using automated solvers. In particular, P4's match-action tables can be treated as *program sketches*—i.e., programs populated with unknown variables called *holes*. The CEGIS loop synthesizes table operations by inductively filling in the program's holes. The controller interacts with these tables incrementally: table entries are usually not changed wholesale, but in small batches. We incrementally synthesize individual control plane operations rather than full tables, which greatly improves Avenir's efficiency.

However, even if one does synthesis incrementally, scaling up to real-world programs remains a significant challenge. Program synthesis has often been used in offline settings, where performance is not a critical concern. However, a typical control plane might modify a table every few milliseconds. To enable online operation, Avenir incorporates heuristic optimizations such as ignoring existing table rules (when possible), and learning "templates" that cache repeated patterns and avoid unnecessary calls to the SMT solver.

Implementation and Evaluation. We have built an implementation of Avenir in OCaml and Z3, and evaluated its effectiveness and scalability. In particular, we used Avenir to perform a "reboot" load test from the ONOS controller with moderate overhead: ONOS takes 15 minutes to generate 40k abstract IPv6 forwarding rules while our tool translates the insertions to a Broadcom pipeline in about 12 minutes. We conducted a series of experiments in which we retarget control planes from one pipeline to another, and show that generated rules successfully forward packets on the Bmv2 software switch. Finally, to assess Avenir's scalability, we ran experiments on synthetic microbenchmarks.

Contributions. This paper presents Avenir, a practical control plane synthesis tool based on the following contributions:

- We present synthesis algorithm that incrementally computes changes to data plane operations, motivated by examples in real-world control planes.
- We formalize our synthesis algorithm and prove (in the appendix) that it is sound and complete.
- We present optimizations that leverage incrementality and domain insights to accelerate synthesis.
- We discuss an implementation and show through case studies and microbenchmarks that Avenir synthesizes control plane operations correctly with modest overheads.

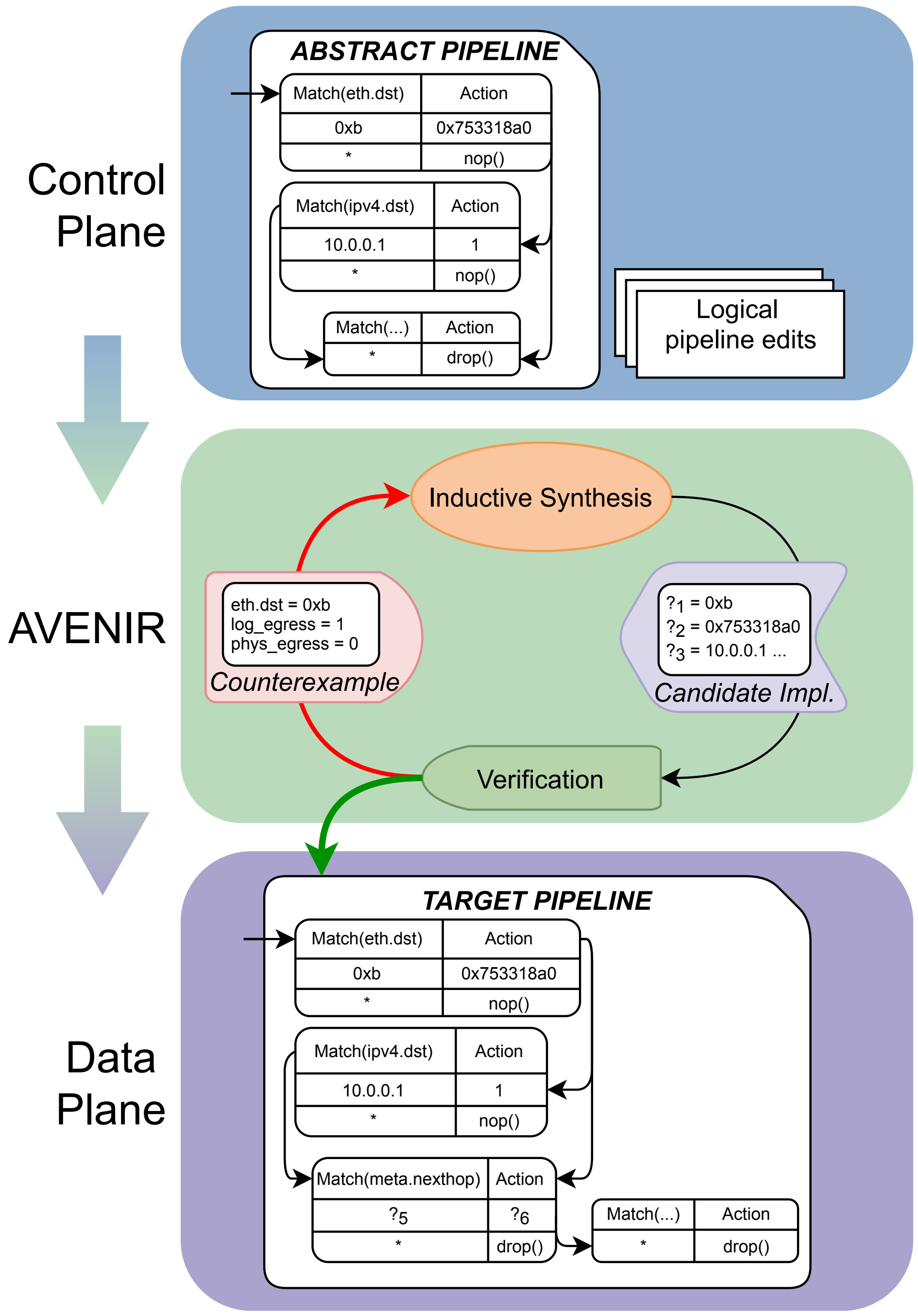


Figure 6.1: Avenir maps control plane operations for an abstract pipeline into corresponding operations for a target using sketch-based synthesis. The synthesis loop alternates between verifying the correctness of a candidate implementation and learning from counterexamples to generate a better one; the holes (e.g., $?_5$) in the target sketch denote missing values that are filled in using an SMT solver.

Pipe₁

```
table l2_fwd {
  key={eth.dst}
  action={set_out}}
table l3_fwd {
  key={ipv4.dst}
  action={set_out}}




apply {
  l2_fwd.apply();
  l3_fwd.apply();

}
```

Pipe₂

```
table l2_fwd {
  key={eth.dst}
  action={set_meta}}
table l3_fwd {
  key={ipv4.dst}
  action={set_meta}}
table lag {
  key={meta};
  action={set_out}
}
apply {
  l2_fwd.apply();
  l3_fwd.apply();
  lag.apply()
}
```

Pipe₃

```
table l2_fwd {
  key={eth.dst}
  action={set_m1}}
table l3_fwd {
  key={ipv4.dst}
  action={set_m2}}
table lag {
  key={m1;m2};
  action={set_out}
}

apply {
  l2_fwd.apply();
  l3_fwd.apply();
  lag.apply()
}
```

OBT

```
  table fwd_table { key={eth.dst;ipv4.dst} action = {set_port} }
  apply { fwd_table.apply() }
```

Figure 6.2: Pipelines used in example scenario.

$(Pipe_1 \Rightarrow Pipe_2)$

```
for each $\rho$ in L2 or L3 do
 if $\rho$.table = $\mathbf{L2\_fwd}$ then
   $\mathbf{L2\_fwd}$.add($\rho$.keys, set_out($\rho$.out))
 else :
   $\mathbf{L3\_fwd}$.add($\rho$.keys, set_meta($\rho$.out))
 fi;
 $\mathbf{LAG}$.add($\rho$.out, set_out($\rho$.out))
```

$(Pipe_1 \Rightarrow Pipe_3)$

```
for each $\rho$ in L2 do
 $\mathbf{L2\_fwd}$.add($\rho$.keys  set_meta($\rho$.out))
 $\mathbf{LAG}$.add(($\rho$.out, $*$), set_out($\rho$.out))
for each $\rho$ in L3 do
 $\mathbf{L3\_fwd}$.add($\rho$.keys  set_meta($\rho$.out))
 $\mathbf{LAG}$.add(($*$, $\rho$.out), set_out($\rho$.out))
 for each $\rho'$ in LAG do
  $\mathbf{LAG}$.add(($\rho'$.m1, $\rho$.out), set_out($\rho$.$out$))
```

Figure 6.3: **The Status Quo** Manual translations from Pipeline 1 to Pipelines 2 and 3. Avenir automates theses translations entirely

6.1 Background and Motivation

As shown in Figure 6.1, Avenir sits between the controller and the data plane, exposing an interface based on an abstract pipeline to the SDN control plane. It intercepts the control operations, translates them to the target pipeline, and passes results to the switch agent to install on the target device. Note that because Avenir

```
$\boxed{(OBT \Rightarrow Pipe_1)}$
for each $\rho$ in $\mathbf{fwd\_table}$ do :
 if $\rho$.ipv4.dst $= *$ then
   $\mathbf{L2\_fwd}$.add($\rho$.ipv4.dst, set_out($\rho$.out))
 elif $\rho$.eth.dst $= *$ then
   $\mathbf{L3\_fwd}$.add($\rho$.eth.dst, set_out($\rho$.out))
 else :
   Failure!
 fi
$\boxed{(OBT \Rightarrow Pipe_2)}$
for each $\rho$ in $\mathbf{fwd\_table}$ do :
 if $\rho$.ipv4.dst $= *$ then
   $\mathbf{L2\_fwd}$.add($\rho$.ipv4.dst, set_out($\rho$.out))
 elif $\rho$.eth.dst $= *$ then
   $\mathbf{L3\_fwd}$.add($\rho$.eth.dst, set_meta($\rho$.out))
   $\mathbf{LAG}$.add($\rho$.out, set_out($\rho$.out))
 else :
   Failure!
 fi
$\boxed{(OBT \Rightarrow Pipe_3)}$
for each $\rho$ in $\mathbf{fwd\_table}$ do :
 if $\rho$.eth.dst $= *$ then
   $\mathbf{L2\_fwd}$.add($\rho$.ipv4.dst, set_out($\rho$.out))
 elif $\rho$.ipv4.dst $= *$ then
   $\mathbf{L3\_fwd}$.add($\rho$.eth.dst, set_meta($\rho$.out))
   $\mathbf{LAG}$.add($\rho$.out, set_out($\rho$.out))
 for each $\rho'$ in LAG do
  $\mathbf{LAG}$.add(($\rho'$.m1, $\rho$.out), set_out($\rho.out$))
 else :
   Failure!
 fi
```

Figure 6.4: **The Status Quo**: Manual translations in pseudocode from "one big table" (OBT) to Pipelines 1 through 3.

works with an abstract notion of a pipeline, it could be used at multiple levels of abstraction—e.g., to implement a driver for a given switch, an abstraction layer like SAI, or even at higher layers of the SDN controller. Likewise, because Avenir operates on switch-by-switch granularity, it can expose different abstract pipelines for different targets. Avenir's synthesis algorithm is sound and its solutions are formally verified, which eliminates the potential for subtle bugs caused by the inherent complexity of the problem, assuming the specifications are correct. Avenir's algorithm is also complete—i.e., given sufficient time, it is guaranteed to find a correct sequence of target operations if it exists.

Status Quo: Manual Control Plane Mappings. Consider a simple running example based on ONOS that illustrates the need for a control plane synthesis tool. Suppose that each switch implements the simple L2-L3 pipeline indicated by Pipe_1 in Figure 6.2. In this pipeline, the output port is set based on the Ethernet and IPv4 destination addresses in the corresponding tables.

As the network matures, its engineers decide to add additional physical data planes—e.g., to incorporate a new generation of hardware or to avoid vendor lock-in. For instance, the pipeline Pipe_2, shown in Figure 6.2, adds a layer of metadata indirection to the physical device to support link aggregation.

To avoid disrupting the control plane, which likely consists of hundreds of thousands of lines of code,[1] the engineers write a *driver* that translates operations written for Pipe_1 into operations for Pipe_2. In this case, the driver, shown on the left of Figure 6.3, is relatively simple: for each rule, it simply copies the output port into meta and inserts a row into the LAG table effectively copying the value

[1]ONOS has currently about 611k lines of Java code [101, 86].

of meta into the output port.

Now, suppose the engineers decide to support a third pipeline ($Pipe_3$ in Figure 6.2), which sets a separate metadata field in each table. The translation (on the RHS of Figure 6.3), is also simple, but requires some care to write—in particular, the L3 table's forwarding decision must always be preferred in the LAG table.

Finally, suppose the engineers want to migrate their original pipeline to a *one big table* abstraction (like OBT in Figure 6.2), similar to OpenFlow. Now, the engineers need to make code changes to all three translations (Figure 6.4).

Of course, the ONOS engineers could compose the translations from the one big table to the first pipeline, and on to the other pipelines. However as more and more logical and physical tables are added, managing a complex cascade of translations would become unwieldy, and hard to maintain.

Control Plane Synthesis with Avenir. Avenir improves upon the state of the art—i.e., writing manual translations—by automating the translation of rules from an abstract pipeline to a target pipeline. Of course, the programmer still needs to write programs that capture the behavior of both pipelines, and that's a non-trivial task. But we believe this should be less challenging than actually writing the translations—akin to describing source and target languages vs. writing a compiler.

To see how this is done, let's explore how Avenir translates abstract $Pipe_1$ L2 insertions into $Pipe_2$ insertions. First, assume, as shown on the left of Figure 6.5, that the L3 table is populated with rules that match on the IPv4 address (10.0.0.1) and set the metadata to (8), and the LAG table matches on that metadata and

$\boxed{Pipe_1}$

L2_fwd

eth.dst	**Action**
ABB28FC	set_out(5)

L3_fwd

ipv4.dst	**Action**
10.0.0.1	set_out(8)

$\boxed{Pipe_2}$ *with holes*

L2_fwd

eth.dst	**Action**
$?_1$	$?_2$

L3_fwd

ipv4.dst	**Action**
10.0.0.1	set_meta(8)
$?_5$	$?_6$

LAG

meta	**Action**
8	set_out(8)
$?_3$	$?_4$

Figure 6.5: Dynamic Configurations used in example scenario. *Pipe*$_2$ is annotated with "holes" to be filled in. During synthesis, Avenir solves for these unknowns and concludes that $?_1 =$ ABB28FC, $?_2 =$ set_meta(5), $?_3 = 5$, $?_4 =$ set_out(5).

forwards out port 8. Consider inserting a single rule into the abstract Pipeline 1 L2 table that matches on eth.dst = ABB28FC and sets the outport to 5. To reflect this update in Pipeline 2, we then need to solve for the unknowns, written as (?) in *Pipe*$_2$ of Figure 6.5. These unknowns model the answers to questions like "Which tables need modification?" and "What should the matches/actions/action data be?"

More formally, the unknowns (?) represent a special kind of variable we instrument our program with, called a *hole*. Programs instrumented with holes are called *sketches*. We heuristically search for a valuation of these holes that makes the behaviors of the two pipelines equivalent. In this example, we could set $?_1 =$ ABB28FC, $?_2 =$ set_meta(5), $?_3 = 5$, and $?_4 =$ set_out(5). Since we do not need to insert a rule in the L3 table, we do not need to find values for these holes. In practice, holes can only be assigned values, not code snippets, like we are doing here for $?_2$ and $?_4$. We will see how to construct these sketches in detail in Section 6.2.2, and we will introduce our synthesis algorithm in Sections 6.2.3, 6.3.1 and 6.4.2.

As a strawman, we might consider an offline approach, where we synthesize the driver code once-and-for-all that translates any abstract operation into equivalent target operations. However, there are many cases (e.g., Figure 6.4) where there is no translation that works for all abstract operations, this synthesis algorithm would fail to produce any solution in many cases where Avenir would succeed. Avenir's online solution allows for a more dynamic and flexible approach.

Incrementality and Optimizations. The key challenge in making Avenir practical is scaling up to handle real-world programs, which typically have at least dozens of tables with thousands of rules. Avenir needs to potentially compute a translation on every abstract control plane operation, so it must be responsive. As another strawman, imagine an approach that computes a full set of table rules on every control plane operation. This strategy might be workable when the tables have only a few rules, e.g., recomputing the existing match in Pipeline 2's L3 table, but it would quickly become a bottleneck if there were say, tens of thousands of rules in L3. Hence, we employ an incremental approach in which we synthesize "deltas" consisting of small batches of control plane operations rather than full tables. By only considering the most recent insertion or deletion into a table, we can often reuse previous solutions and avoid redundant recomputation.

Going a step further, we can cache "templates" derived from previous solutions to help translate future operations. For example, on the next insertion into L2, we can try to reuse the same stucture by inserting into L2_fwd and LAG, with actions set_meta and set_out, forcing the argument to set_meta to equal the LAG table match.

6.2 Control Plane Synthesis

Our synthesis algorithm is based on CEGIS [104]. The core of CEGIS is a loop with two main components: verification and inductive synthesis. In each iteration of the loop, a candidate implementation is run through the verification component to check correctness. If verification fails, a counterexample trace is produced, allowing the inductive synthesis component to learn from this failure to generate a better candidate. The loop terminates when verification ultimately succeeds.

In our setting, the CEGIS loop is run for each insertion into the abstract pipeline. Inductive synthesis produces candidate control plane implementations for the target pipeline, and verification checks whether the behavior of the two pipelines are equivalent. The rest of this section discusses the CEGIS components in detail. Section 6.3 discusses optimizations that make this approach efficient and scalable.

6.2.1 Basic Definitions and Verification

The *verification* component of the CEGIS loop determines whether the synthesized control plane operations implement the same packet-processing behavior on the target pipeline as on the abstract pipeline. We model packets as finite maps from a fixed set of header and metadata fields to bit vectors, and say two packets are equal and write $pkt = pkt'$ if they agree on all header fields. Packets have a direct interpretation as a boolean formula: for headers Hdr and a list $\overline{x} \subseteq \mathsf{Hdr}$, we write $pkt[\overline{x}]$ to mean $\bigwedge_{x \in \overline{x}} x = pkt.x$.

$$
\begin{array}{rcll}
c & ::= & & (c \in \mathrm{GPL}) \\
 & \mid & f := e & \textit{Assignment(*)} \\
 & \mid & c; c & \textit{Sequence(*)} \\
 & \mid & \mathsf{if}\ \overline{b \to c}\ \mathsf{fi} & \textit{Guarded Commands(*)} \\
 & \mid & \mathsf{apply}\ t & \textit{Table Application} \\
a & ::= & & (a \in \mathsf{Act}) \\
 & \mid & \lambda\ (\overline{x}).\ c(*) & \textit{Function} \\
t & ::= & \left\{ \begin{array}{l} \mathit{name} : \mathsf{Func}; \\ \mathit{keys} : \mathsf{Func}^{+}; \\ \mathit{actions} : \mathsf{Act}^{+}; \\ \mathit{default} : \mathsf{Act} \end{array} \right\} & \textit{Table Definition} \\
\delta, \varepsilon & ::= & & (\delta \in \mathsf{Edit}) \\
 & \mid & \mathsf{A}_x(\rho) & \quad \textit{Insertion} \\
 & \mid & \mathsf{D}_x(n) & \quad \textit{Deletion} \\
\rho & \in & \mathsf{Row} = \mathsf{List}[2^*] \times \mathsf{List}[2^*] \times \mathbb{N} & \\
\tau, \sigma & \in & \mathsf{Inst} = \mathsf{Func} \Rightarrow \mathsf{List}[\mathsf{Row}] & \\
v & ::= & [n]_n & \textit{Bitvector } (v \in 2^*) \\
h & ::= & \left\{ \begin{array}{l} \mathit{name} : \mathsf{Func}; \\ \mathit{width} : \mathbb{N} \end{array} \right\} & \textit{Header Field } (h \in \mathsf{Hdr}) \\
m & ::= & \left\{ \begin{array}{l} \mathit{name} : \mathsf{Func}; \\ \mathit{width} : \mathbb{N} \end{array} \right\} & \textit{Metadata Field } (h \in \mathsf{Meta}) \\
f & \in & \mathsf{Hdr} \cup \mathsf{Meta} & \\
x & \in & \mathsf{Func} & \\
n & \in & \mathbb{N} &
\end{array}
$$

Figure 6.6: Pipeline syntax. Actions vary under starred variants

Syntax and Semantics. In Figure 6.6, we define the syntax of pipelines, Notice that we have departing from our very abstract GPL formalism from earlier chapters, to one that adds more of the P4-style surface syntax. We have also done away with nondeterministic choice, opting instead for concrete conditionals. For Avenir, a pipeline program is a just a command $c \in \mathrm{GPL}$, that denotes a packet processing function, which we write $[\![c]\!] : \mathsf{Packet} \to \mathsf{Packet}$.

There are a few ways to compositionally build a pipeline program. First, fields

$f \in \mathsf{Var}$ can be assigned values via the command $f := e$, which updates the packet pkt to $pkt\{f \mapsto n\}$, where e evaluates to n in pkt. Further, commands can be sequenced, $c_1; c_2$, which first executes c_1 then c_2. We can also use conditional control flow, written $\mathsf{if}\ b_1 \to c_1\ \ldots\ b_n \to c_n\ \mathsf{fi}$, which executes command c_i on the incoming packet pkt for the smallest-indexed b_i that evaluates to true on pkt. These conditionals are similar to Dijkstra-style guarded commands [32].

Finally, table application $\mathsf{apply}(t)$ executes match-action table t. Tables are represented as records, where $t.name$ is table's name; $t.keys$ is a list of packet headers referred to by name; $t.actions$ is the list of actions (which are lexically-scoped anonymous functions $\lambda(\overline{x}).c$); and $t.default$ is the command that is executed when the table is missed. Only certain commands c may occur inside an action (denoted with a $(*)$ in Figure 6.6)—e.g., table application is not allowed.

Our configurations are constrained to have a specific data structure, rather than arbitrary formulae. This will enable us to carefully structure our synthesis algorithm in the subsequent sections. To represent entries in a table t, we maintain a *table instantiation* $\tau : \mathsf{Func} \to \mathsf{List}[\mathsf{Row}]$, alongside the syntactic pipeline, which maps table names to their row lists. We write Inst for the set of all instantiations. We refer to the pair (c, τ) as the *pipeline state*. A row $\rho \in \mathsf{Row}$ is a triple $\rho = (\overline{m}, \overline{d}, a)$, where $\overline{m}$ are matches, a is the action index and $\overline{d}$ is the action data.

We can define a source-to-source syntactic transformation $c[\tau]$ that replaces every occurence of $\mathsf{apply}(t)$ in c with a guarded command encoding the rows of the table $\overline{\rho} = \tau(t.name)$, as follows, where the ith row $\rho_i = (\overline{m_i}, \overline{d_i}, a_i)$:

$$\textsf{if} \begin{pmatrix} t.keys = \overline{m_0} & \rightarrow & t.action[a_0](\overline{d_0}) \\ & \cdots & \\ t.keys = \overline{m_n} & \rightarrow & t.action[a_n](\overline{d_n}) \\ \textsf{true} & \rightarrow & t.default \end{pmatrix} \textsf{fi}$$

We say that a row $(\overline{m}, \overline{d}, a)$ is well-defined for a table t when $|\overline{m}| = |t.keys|$, $a < |t.actions|$, and for the parameters $\overline{x}$ of $t.actions[a]$, $|\overline{d}| = |\overline{x}|$. Further, an instance is well-defined when all of its rows are well-defined for their tables, and a command is well-defined when no two tables have the same name. We assume that commands and instantiations are well-defined, and that there are no bit-width mismatches: both are easy to check statically.

Finally, we have control plane edits ($\delta \in \textsf{Edit}$), which are operations that allow the control plane to modify table instantiations. We interpret them as functions, writing $\delta(\tau) \in \textsf{Inst}$. There are two kinds of edits: insertions and deletions. For a given instance τ, an insertion $\textsf{A}_x(\rho)(\tau)$ appends ρ to the end of $\tau(x)$ (meaning it has the lowest priority). If $\tau(x)$ has a row ρ' with the same matches as ρ, the inserted row is dropped. A deletion $\textsf{D}_x(i)(\tau)$ removes the ith element from $\tau(x)$.

Now that we know how to interpret configured pipelines as functions, we say $c_1 = c_2$ when they are functionally equivalent , concretely $\forall pkt.\ [\![c_1]\!]\ pkt = [\![c_2]\!]\ pkt$. To check this condition, we use predicate transformer semantics to generate a verification condition [47], written $c_1 \equiv c_2$, which we check using an SMT solver, by running $\textsf{CheckSat}(c_1 \not\equiv c_2)$. If the solver returns $\textsf{UNSAT}$, we conclude the programs are equivalent. Otherwise, it returns $\textsf{SAT}$ as well as a model that encodes a counterexample χ—i.e., an input and output packet pair χ that demonstrates different behavior in the abstract and physical programs, writing χ_0 and χ_1 for the

HOLE	DESCRIPTION
$?\mathsf{Del}_{\mathsf{t,i}} = 1$	Delete row i in table t
$?\mathsf{Add}_{t,j} = 1$	Add j rows to table t
$?\mathsf{Act}_{t,j} = i$	New Row j in table t (if added), has action i
$?\mathsf{k}_{t,j} = v$	New Row j in table t (if added), matches header k with v
$?\mathsf{d}_{t,j,i} = v$	New Row j in table t (if added with action i) has action data for parameter d set to v

Figure 6.7: Summary of holes used in sketching.

input and output packets respectively. It is easy to prove that this validity check implies functional equivalence.

Theorem 6.2.1. *For every pair of pipelines c_1, c_2, if $c_1 = c_2$ then* $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{UNSAT}$*, and if $c_1 \neq c_2$ then* $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{Sat}\ \chi$.

Proof. By soundness of verification conditions with respect to the denotational semantics of guarded commands [32, 47]. □

6.2.2 Synthesizing Candidates via Sketches

To propose new candidate programs for verification, we use a technique called Sketching [105]. A *sketch* is a command containing special variables called *holes*. Aside from holes for values (i.e., ?k for match keys and ?d for action data), which we introduced in Section 6.1, we also need holes for table entries, corresponding to deletions (?Del), insertions (?Add) and action choice (?Act). The meaning of these holes is described in Figure 6.7.

To compute a candidate solution in our CEGIS loop, we first instrument the program with holes. We write $instr(c, \tau, \overline{\delta}, n)$ to describe the program $\overline{\delta}(\tau(c))$ with deletion holes for every row in τ, and holes for n row insertions. We do not add deletion holes for insertions in $\overrightarrow{\delta}$, which is crucial for the completeness theorem (Section 6.3). We lift this function from tables to programs in the obvious way.

Consider the $\mathsf{L2}$ table from pipelines 1 and 2. To instrument it with holes, allowing for a single insertion, we would insert a deletion hole for the existing rule and a single row of insertion holes, yielding the following sketch:

	Match(eth.dst)	**Action**
$?\mathsf{Del} = 0$	ABB28FC	$\mathsf{set\_out}(5)$
$?\mathsf{Add} = 1$	$?\mathsf{eth.dst}$	$\mathsf{if}\ ?\mathsf{Act} = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p})$ $?\mathsf{Act} = 1 \rightarrow \mathsf{drop}()$ fi

A possible model for these holes that matches the destination MAC address with $\mathsf{00:00:00:00:00}$ and drops the packet, is $\{?\mathsf{Del} \mapsto 0, ?\mathsf{eth.dst} \mapsto 0, ?\mathsf{Act} \mapsto 1\}$. Note that $?\mathsf{p}$ is irrelevant, so we omit it from the model.

Of course, sketches represent a vast search space of edits: every existing table row can be deleted, and up to n rows can be inserted. Blindly searching through this space would not scale in practice. Instead, we learn from counterexamples to help guide the search toward a solution.

6.2.3 Counterexample-Guided Search

When the solver determines that a proposed candidate pipeline is not equivalent to the abstract pipeline, it generates a counterexample χ that encodes an input-output packet pair. This pair corresponds to a behavior of the abstract switch that is not replicated in the candidate or vice versa. We can use this counterexample to

$$\mathsf{fix\_cex}(p, \sigma, \chi, n, \overline{x}) \triangleq \chi_0[\overline{x}] \Rightarrow wp\,(instr(p, \sigma, \cdot, n), \chi_1[\overline{x}])$$
$$\mathsf{model}(p, \sigma, X) \triangleq \mathsf{CheckSat}\left(\forall \overline{x}.\ \textstyle\bigwedge_{\chi \in X} \mathsf{fix\_cex}(p, \sigma, \chi, |X|, \overline{x})\right)$$

Figure 6.8: The model function. In the above, the vector $\overline{x}$ is all of the non-hole variables that occur in the formula.

```
cegis($l, p, \tau, \sigma, \overline{\delta}, X$) $\triangleq$
  match CheckSat($l[\tau] \not\equiv p[\overline{\delta}(\sigma)]$) with
    | UNSAT $\rightarrow$ Ok $\overline{\delta}$
    | SAT $\rightarrow$
        match model($p, \overline{\delta}(\sigma), \{\chi\} \cup X$) with
          | UNSAT $\rightarrow$ Fail
          | SAT $\overline{\delta'}$ $\rightarrow$ cegis($l, p, \tau, \sigma, \overline{\delta'}, \{\chi\} \cup X$)
```

Figure 6.9: Simple Algorithm for Control Plane Synthesis.

guide our search. More formally, we use the weakest precondition $wp(c, \varphi)$ whose satisfying models are inputs that, after executing c, yield outputs satisfying φ.

The fix_cex function constructs the formula $\chi_0[\overline{x}] \Rightarrow wp(s, \chi_1[\overline{x}])$ for the sketch $s = instr(p, \sigma, \cdot, |X|)$. The formula identifies edits that when applied to the physical pipeline state (p, σ) produce the input-output behavior indicated by χ.

The function model in Figure 6.8 lifts fix_cex over all counterexamples X that have been seen so far. Notice that we only instrument the physical pipeline with $|X|$ insertion holes since each counterexample hits at most one rule in each table.

6.2.4 Synthesis Algorithm

The full synthesis algorithm is presented in Figure 6.9. Given a abstract pipeline l, a target pipeline p, an abstract table instantiation τ, a target table instantiation σ, a sequence of physical edits $\overline{\delta}$, and a set of counterexamples X, cegis$(l, p, \tau, \sigma, \overline{\delta}, X)$

L2_fwd =

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)

$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow$ set_out$(?\mathsf{p}_0)$
		$?\mathsf{Act}_0 = 1 \rightarrow$ drop()
		fi

$?\mathsf{Add}_1 = 1$	$?\mathsf{eth.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow$ set_out$(?\mathsf{p}_1)$
		$?\mathsf{Act}_1 = 1 \rightarrow$ drop()
		fi

L3_fwd =

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
$?\mathsf{Del}_1 = 0$	8.8.8.8	set_out(47)

$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_2 = 0 \rightarrow$ set_out$(?\mathsf{p}_2)$
		$?\mathsf{Act}_2 = 1 \rightarrow$ drop()
		fi

$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_3 = 0 \rightarrow$ set_out$(?\mathsf{p}_4)$
		$?\mathsf{Act}_3 = 1 \rightarrow$ drop()
		fi

Figure 6.10: Basic Sketch for $Pipe_1$: Satisfiable for packets that hit L2's first row and L3's second.

L2_fwd =

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)

$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow$ set_out$(?\mathsf{p}_0)$
		$?\mathsf{Act}_0 = 1 \rightarrow$ drop()
		fi

L3_fwd =

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
	8.8.8.8	set_out(47)

$?\mathsf{Add}_1 = 1$	$?\mathsf{ipv4.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow$ set_out$(?\mathsf{p}_1)$
		$?\mathsf{Act}_1 = 1 \rightarrow$ drop()
		fi

Figure 6.11: Incremental Sketch for $Pipe_1$: Unsatisfiable for packets that hit L2's first row and L3's second, which triggers backtracking, remembering that the previously-synthesized edit was incorrect.

produces a sequence of edits $\overline{\varepsilon}$ such that $l[\tau] = p[\overline{\varepsilon}(\sigma)]$ if one exists. We initially call the algorithm with $\overline{\delta} = []$ and $X = \{\}$. First, we call the SMT solver to check for equality. If the programs are equal, we are done, and return $\overline{\delta}$. Otherwise, we get a counterexample χ and solve for new edits by augmenting X with χ, applying the edits to the target pipeline and calling model. If it returns UNSAT, there is no way to make the programs equivalent and we fail. Otherwise, we get a new sequence of edits and keep searching.

6.2.5 Formal Properties

Next we establish two formal properties for our synthesis algorithm: soundness and completeness. Soundness means that synthesized target operations produce equivalent behavior.

Theorem 6.2.2 (Soundness). *For every $l, p \in \mathrm{GPL}$, $\tau, \sigma \in \mathsf{Inst}$, $\overline{\delta} \in \mathsf{List}[\mathsf{Edit}]$, and $X \subseteq [\![l[\tau]]\!] \cap [\![p[\overline{\delta}(\sigma)]]\!]$ if $\mathsf{cegis}(l, p, \tau, \sigma, \overline{\delta}, X) = \mathsf{Ok}\ \overline{\varepsilon}$ then $l[\tau] = p[\overline{\varepsilon}(\sigma)]$.*

Proof. Follows from Theorem 6.2.1. □

Completeness says that if a solution exists, then our synthesis algorithm will (eventually) find it.

Theorem 6.2.3 (Completeness). *For every $l, p \in \mathrm{GPL}$, $\tau, \sigma \in \mathsf{Inst}$, $\overline{\delta} \in \mathsf{List}[\mathsf{Edit}]$, and $X \subseteq [\![l[\tau]]\!] \cap [\![p[\overline{\delta}(\sigma)]]\!]$, if $\exists \overline{\delta'} \in \mathsf{List}[\mathsf{Edit}].\ l[\tau] = p[\overline{\delta'}(\sigma)]$ then $\exists \overline{\delta''} \in \mathsf{List}[\mathsf{Edit}].\ \mathsf{cegis}(l, p, \tau, \sigma, \overline{\delta}, X) = \mathsf{Ok}\ \overline{\delta''}$ and $l[\tau] = p[\overline{\delta''}(\sigma)]$.*

Proof. By induction on the size of $\mathsf{Packet} \setminus \pi_1(X)$. □

Limitations The main limitation of this first synthesis algorithm is that the number of queries is bounded by the number of counterexamples—i.e., every possible packet. Given an MTU of n, there could be as many as 2^n packets.

6.3 A Scalable Solution: Incremental Synthesis

To obtain a scalable synthesis algorithm, we first exploit the insight that the control plane operates in an incremental fashion—i.e., before each control plane operation, the data planes are already equivalent, so we only need to handle incremental changes to the abstract program, such as adding or deleting a rule. In the common case, we do not have to resynthesize all of the previously generated rules. However, some care is needed as certain control plane operations do require deleting previously installed rules.

6.3.1 Single Counterexample-Guided Search

Our first enhancement to the basic synthesis algorithm is to only add insertion holes to solve for the most recent counterexample, and only add deletion holes for state that existed before synthesis began, which greatly reduces the number of holes we need to produce as we explore the space. Instead of instrumenting the program with insertion holes for every counterexample, we only do it for the most recent one.

Consider again the L2 and L3 tables from $Pipe_1$ with the initial state depicted in Figure 6.5. We want to synthesize edits that send Ethernet packets that miss in the L2_fwd with destination DECAFBAD out on port 47. Suppose the first counterexample has input packet $\chi_0 = \{\mathsf{eth.dst} \mapsto \mathsf{DECAFBAD}, \mathsf{ipv4.dst} \mapsto \mathsf{8.8.8.8}\}$, and output packet $\chi_1 = \chi_0\{\mathsf{out} \mapsto \mathsf{47}\}$. Let's say on the first iteration we produce the (incorrect) edit to $\mathsf{L2\_fwd}$ that maps $\mathsf{ipv4.dst} = 8.8.8.8$ to $\mathsf{set\_out}(47)$, and the verification step will provide a new counterexample.

Suppose the next counterexample has input packet $\chi_0' = \{\mathsf{eth.dst} \mapsto \mathsf{ABB28FC}, \mathsf{ipv4.dst} \mapsto \mathsf{8.8.8.8}\}$, and output packet $\chi_1' = \chi_0'\{\mathsf{out} \mapsto 5\}$. Now the simple algorithm will produce the sketch in Figure 6.10, which can be solved by deleting the already inserted row ($?\mathsf{Del}_1 = 1$) and adding the single required row to the L2 table ($?\mathsf{Add}_0 = 1$, $?\mathsf{eth.dst}_0 = \mathsf{DECAFBAD}$, $?\mathsf{Act}_0 = 0$, $?\mathsf{p}_0 = 47$, and remaining $\mathsf{Add}/\mathsf{Del}$ holes disabled).

In contrast, the incremental search will first create the unsatisfiable sketch shown in Figure 6.11. There is no way to fill its holes to satisfy the above counterexample. We backtrack with the knowledge that $?\mathsf{ipv4.dst} \neq \mathsf{8.8.8.8}$ and attempt to solve the original sketch with respect to the original counterexample, and the only remaining solution is correct.

First, notice that the final simple sketch uses 21 holes, whereas each incremental sketch uses only 10. On the other hand, the incremental search sends 3 sketches to the solver as opposed to the simple search, which only sends 2. Why do we want to send *more* queries to Z3 instead of less? This is a result of the NP-completeness of SAT/SMT solving. Solving more formulae with fewer variables is often faster than solving fewer formulae with more variables. Here, the search space size for the 3 incremental sketches is approximately $3 \cdot |B|^{10}$, whereas for "simple" query it is approximately $|B|^{21}$, where $|B|$ is the size of the bitvector domain.

Further, observe that the incremental sketches we send will *always* have 10

$$\mathsf{fix\_cex}(p,\sigma,\overline{\delta},\overline{x},\chi) \triangleq pkt[\overline{x}] \Rightarrow wp(instr(p,\sigma,\overline{\delta},1), pkt'[\overline{x}])$$

$$\mathsf{model'}(p,\sigma,\overline{\delta},\chi,\varphi) \triangleq \mathsf{SAT}\begin{pmatrix} \forall\overline{x}.\mathsf{fix\_cex}(p,\sigma,\overline{\delta},\overline{x},\chi)) \\ \wedge\ \varphi \wedge \mathsf{HEURISTIC}() \end{pmatrix}$$

Figure 6.12: The model′ function computes edits to physical state (p,σ) to accomodate the counterexample χ. The oracle soundly restricts the search space.

```
$\mathsf{cegis}(l,p,\tau,\delta,\sigma) \triangleq \mathsf{verify}(l,p,\delta(\tau),\sigma,[\,])$
$\mathsf{verify}(l,p,\tau,\sigma,\overline{\delta}) \triangleq$
  match $\mathsf{CheckSat}(l[\tau] \not\equiv p[\overline{\delta}(\sigma)])$ with
    | UNSAT $\rightarrow$ Ok $\overline{\delta}$
    | SAT $\chi \rightarrow \mathsf{solve}(l,p,\tau,\sigma,\overline{\delta},\chi,\mathsf{true})$
$\mathsf{solve}(l,p,\tau,\sigma,\overline{\delta},\chi,\varphi) \triangleq$
  match $\mathsf{model'}(p,\sigma,\overline{\delta},\chi,\varphi)$ with
      | UNSAT $\rightarrow$ Fail
      | SAT $\overline{\delta'} \rightarrow$
          match $\mathsf{verify}(l,p,\tau,\sigma,\overline{\delta}\circ\overline{\delta'})$ with
            | Ok $\overline{\delta''} \rightarrow$ Ok $\overline{\delta''}$
            | Fail $\rightarrow \mathsf{solve}(l,p,\tau,\sigma,\overline{\delta},\chi,\varphi \wedge \neg\overline{\delta'})$
```

Figure 6.13: The incremental backtracking CEGIS algorithm.

holes, independent of the number of counterexamples, whereas the simple sketch will continue to add holes as the number of counterexamples grows.

We formalize this new incremental model-finding function model′ in Figure 6.12. It is defined in term of a satisfiability check for a conjunction of three sub-formulas. The first conjunct uses a modified fix_cex function that instruments the program with one addition hole per table. The second conjunct, φ, limits the search by preventing models from reoccurring. The final conjunct is a search oracle HEURISTIC() that computes restrictions on the search space. The only constraints on HEURISTIC() are that it must not add covered rules or previously-deleted rules (to avoid looping), and it must not permanently preclude any solution (to ensure completeness).

6.3.2 Incremental Synthesis Algorithm

We present our incremental synthesis algorithm in Figure 6.13. It comprises two mutually recursive functions: verify, which checks the verification condition and solve, which generates new models. Both functions take the same arguments: the abstract and target programs and instantiations ((l, τ) and (p, σ) respectively), and a sequence of edits to the target program $\overline{\delta}$. They either return Ok $\overline{\delta}'$, where $\overline{\delta}$ is the prefix of $\overline{\delta}'$ and $\mathsf{CheckSat}(l[\tau] \not\equiv p[\overline{\delta}'(\sigma)]) = \mathsf{UNSAT}$, or Fail, if there is no such $\overline{\delta}'$. The cegis function is the "main" method. It takes the abstract and target pipelines (l and p) and instantiations (τ and σ) as arguments, as well as the abstract edit δ. It then applies δ to τ and invokes verify with no target edits.

The verify function resembles the cegis function of Section 6.2. It first checks whether the programs are equal, and if so returns Ok $\overline{\delta}$. Otherwise it calls solve with an initial counterexample χ and an unrestricted model, which searches for an edit to make the programs equivalent.

The solve function takes the standard arguments, with the addition of a counterexample χ and the model space restriction formula φ, which keeps track of failed solutions for χ, to prevent repetition. First, model′ searches for a target edit that corrects the behavior for the counterexample. If none exists, we return Fail, indicating that there is no sequence of equivalent target edits with the prefix $\overline{\delta}$. Otherwise, model′ provides a model $\overline{\delta'}$. In this case we extend the running sequence of edits to $\overline{\delta} \circ \overline{\delta'}$ and call back to verify. If successful, we return the result, otherwise we preclude $\overline{\delta'}$ from the space of possible models φ (writing $\neg\overline{\delta'}$ for the negation of valuations that produce $\overline{\delta'}$.) Then we recursively call solve and continue searching within this restricted space of models.

6.3.3 Formal Properties

We prove that the incremental algorithm is also sound and complete. As with the simpler algorithm, the proof of soundness follows by the correctness of the verification condition.

Theorem 6.3.1 (Incremental Soundness)**.** *For every* $l, p \in \text{GPL}$*,* $\tau, \sigma \in \mathsf{Inst}$*,* $\delta \in \mathsf{Edit}$*,* $X \subseteq [\![l[\tau]]\!] \cap [\![p[\overline{\delta}(\sigma)]]\!]$*, if* $\mathsf{cegis}(l, p, \tau, \sigma, \delta, X) = \mathsf{Ok}\ \overline{\varepsilon}$*, then* $l[\tau] = p[\overline{\varepsilon}(\sigma)]$*.*

Proof. Again, the result follows from Theorem 6.2.1. □

As in the simple synthesis algorithm, incremental completness relies on the finite domain, which here is the product of two finite domains: (1) sequences of reachable edits that do not redundantly add and delete a rule, and (2) the number of valuations for the holes introduced by the instr function.

Theorem 6.3.2 (Incremental Completeness)**.** *For every abstract program* l*, target program* p*, abstract instantiation* τ*, target instantiation* σ *and abstract edit* δ *if* $\exists \overline{\varepsilon} \in \mathsf{List[Edit]}.\ l[\delta(\tau)] = p[\overline{\varepsilon}(\sigma)]$ *then* $\exists \overline{\delta'} \in \mathsf{List[Edit]}.\ \mathsf{cegis}(l, p, \tau, \delta, \sigma, [\,]) = \mathsf{Ok}\ \overline{\delta'}$ *and* $l[\delta(\tau)] = p[\overline{\delta'}(\sigma)]$*.*

Proof. By strong outer induction on the size of the reachable non-deleting edit sequences, and strong inner induction on the (lexicographically ordered) size of the counterexample set and the number of models in each model space. □

Theorem 6.3.2 proves that Avenir translates abstract operations given unbounded resources. In practice, Avenir's effectiveness relies on heuristics and optimizations.

6.4 Heuristics and Optimizations

Avenir offers a number of heuristic optimizations designed to help it scale to larger networks. Interestingly, these optimizations need not be sound. We introduce a run-time check for soundness and revert the optimization if it fails. We focus on two classes of optimizations: verification and model finding.

6.4.1 Exploiting Incrementality

The key to scalable synthesis is to adopt an incremental approach and focus on edits, while incorporating further optimizations within the verification and synthesis steps.

Fast Counterexamples. In the incremental setting, we know that a new abstract insertion δ must be the cause of any semantic difference with the target pipeline. We symbolically compute packets that hit δ via an SMT query that gives us a potential counterexample packet *pkt*. We use the denotational semantics to check whether *pkt* is a real counterexample. If *pkt* doesn't induce different behavior we retry the query (in practice 10 times) until we either obtain a true counterexample, or resort back to the standard equivalence check.

Program Slicing. We leverage the incrementality assumption to use program slicing to verify only the part of the program that changes. This isn't always sound, so we check that the abstract edits are reachable iff the target edits are. We also have a faster and stronger constraint that checks that the abstract and target matches are disjoint from the extant rules. If both conditions fail, we run the full equivalence check. In practice, slicing composes with constant propagation and dead code elimination to normalize the queries.

Query Templates. The queries produced using program slicing are often syntactically similar. So when we see two validity queries that only differ in their specific concrete values, we try to abstract those concrete values into a universally quantified variable. We then check whether that more-general query is valid. If it is, we add it to a cache of templates, otherwise we continue in a CEGIS loop by negating the valuation of the quantified variables and trying again. Whenever we get future queries that are instances of the template, we can return VALID without having to consult the SMT solver.

Translation Templates. As with queries, we can cache translations of operations by generalizing over their concrete values to obtain a template. The template observes the way that concrete values are mapped from previously-seen abstract insertions into their equivalent target insertions, and structurally replicates that mapping on the new abstract insertion. It also observes the cache of translations for differing constants and generates unused constants for new rules which optimizes for metadata patterns like in $Pipe_2$ and $Pipe_3$ from Figure 6.2. Note that before adding a solution to the cache, Avenir optionally reduces its size, by heuristically removing superfluous target edits, which improves the generality of the solution. When no template applies, Avenir relies on a heuristic-guided search.

6.4.2 Model-Finding Heuristics

Now we describe the implementation of the $\mathsf{HEURISTIC}()$ oracle, which abstracts a combination of heuristics. In our formalization, we assume that the heuristics are always complete. However in practice, many of Avenir's individual heuristics are not; when a given combination fails, we disable some and try again with a different combination. This search through the heuristics is currently hard-coded, but we plan to support user control of the search strategy and custom heuristics. We describe the heuristics useful in our experiments here.

Ternary and Optional Matching. In the previous sections, we only inserted holes to generate exact matches. We can generate ternary matches for a match key k, which allows us to represent, say, a wild-carded IPv4 source address in only a single row (rather than 2^{32} exact-match rows). To do this, we generate a pair of holes $?\mathsf{k}$ and $?\mathsf{k}_{\mathtt{mask}}$ and encode the match as $\mathsf{k}\&?\mathsf{k}_{\mathtt{mask}} = ?\mathsf{k}$. To eliminate duplicate keys we also enforce the constraint $?\mathsf{k}\&?\mathsf{k}_{\mathtt{mask}} = ?\mathsf{k}$. For optional matches, we restrict the masks to be all 1s or all 0s.

Exact and Mask Hints. When a row is inserted into the abstract pipeline, the non-wildcarded keys K of that row are likely relevant in classifying packets. So, we force the relevance of matches on fields in K, either by copying the abstract match values into the target edits (which is very optimistic), or by forcing their masks (if masking is enabled) to be all 1s.

Action Hints. Given a counterexample (pkt_0, pkt_1), we can observe the variables that change in the abstract program, i.e., $\Delta = \{x \mid pkt_0.x \neq pkt_1.x\}$, and ensure that all edits have actions that can influence the value of some variable in Δ.

Other Optimizations. Our final collection of optimizations are based on intuitive heuristics that arise often in practice.

- **Reachable Adds.** We force synthesized models to be reachable using the counterexample driving the search.
- **Prefer Adds.** We try to find a solution that does not require deleting existing rules.
- **Prefer Non Zero Models.** We enforce $?\mathsf{k} \neq 0 \neq ?\mathsf{d}$ for all key and data holes, unless they are wildcarded.
- **Bounded Edits.** We restrict the search space so that backtracking is triggered beyond specified limits.
- **Previous Counterexamples.** We try to synthesize rules that do not violate previously-seen counterexamples.

6.5 Implementation

We implemented Avenir in approximately 11K lines [101] of OCaml code that interfaces with Z3 [31]. Our implementation accepts a description of an abstract and a target pipeline, sequences of insertions to both programs (to construct the initial state), as well as a sequence of abstract edits to synthesize. Avenir then produces a sequence of edits to the target program (or fails if no such sequence exists). All of the optimizations described in 6.4 are configurable as command line flags. In our implemention, we use an efficient encoding of the weakest precondition [47], which has linear size for the programs in our internal syntax.

P4 Program Encoding. The front-end of our implementation supports a large subset of P4, via an encoding from P4's control blocks into Avenir's internal syntax. This translation resembles previous work on verifying P4 programs [72]. Of course, P4 is a larger language than Avenir's syntax. We support more complex P4 language constructs by desugaring them into sequences of internal commands.

We currently assume that all of the data plane programs use the same parser and headers. Hence, in cases where a mapping only exists due to invariants enforced by the parser—e.g., that a packet cannot simultaneously have IPv4 and IPv6 headers—these assumptions must be manually encoded as annotations. We also ignore match kinds and assume all matches are either exact, ternary or optional, depending on command line flags. Finally, we manually encode certain device-specific behaviors such as the initial value fields and the drop port value. Our implementation is on GitHub[2] under an open-source license.

6.6 Evaluation

To evaluate Avenir, we demonstrate its functionality under a variety of synthetic and realistic scenarios, and measured its performance against hand-written baselines. First, we show how Avenir can automatically retarget a given abstract pipeline to multiple target pipelines (Section 6.6.1). Second, we pass packets through the Bmv2 software switch using the generated rules, which both shows

[2]Available at `https://github.com/cornell-netlab/avenir`

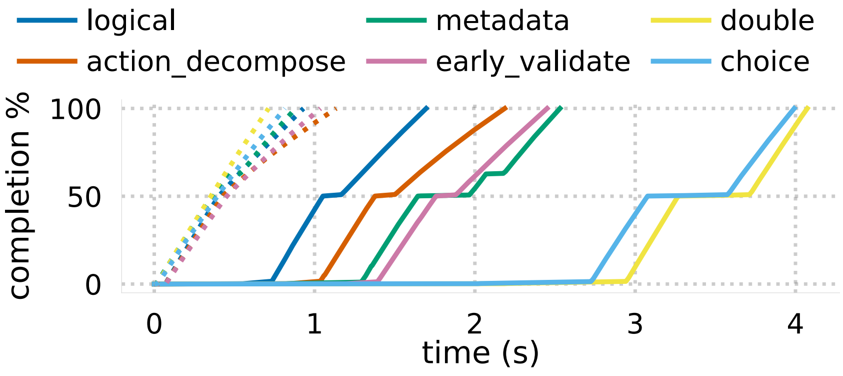


Figure 6.14: Retargeting case study: solid lines show cold-start completion %; dotted lines show hot-start completion %.

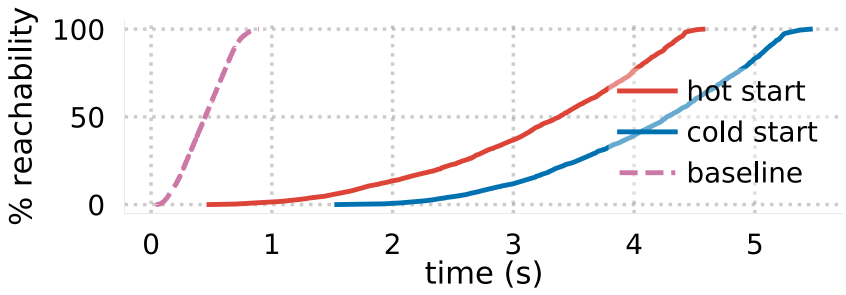


Figure 6.15: Proportion of all pairs of 64 hosts connected in a star topology that have completed a successful IPv4 ping.

they are correct and quantifies Avenir's performance when installing rules for multiple hosts (Section 6.6.2). Third, we present a case study consisting of a realistic workload drawn from the Trellis data center fabric, running on top of the ONOS SDN controller [17] (Section 6.6.3). Finally, we study Avenir's scalability via a suite of microbenchmarks (Section 6.6.4). Our evaluation pays particular attention to the caches, as these are particularly important to obtain good performance.

Summary of Results. Overall, our evaluation shows that, in a variety of cases, Avenir can translate large numbers of rules efficiently. The retargeting, emulation, and ONOS experiments show that Avenir is effective at mapping to and from a variety of programs, and demonstrate that the caching optimizations are highly

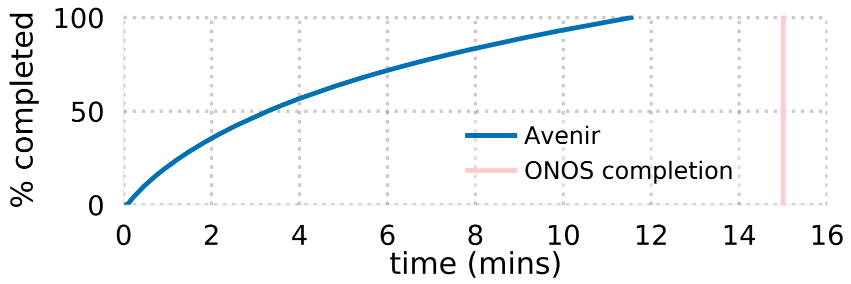


Figure 6.16: Completion graph for mapping 40k fabric.p4 IPv6 route insertions onto bcm.p4; ONOS takes around 15 min.

effective at reducing overheads.

6.6.1 Retargeting Study

Avenir allows operators to expose a single pipeline abstraction to the control plane, while implementing the forwarding logic over a myriad of physical devices. We demonstrate this use case via a retargeting study, where we retarget an initial program onto a variety of different target pipelines.

The logical program logical.p4 is a simple L2-L3 pipeline followed by a PUNT table that performs packet validation on all headers and metadata. We describe 5 additional target pipelines in terms of the changes to logical.p4:

(early_validate.p4) Replaces the PUNT table of logical.p4 with an ACL that can only match on addresses. Adds a validation table prior to the L2 table that matches on the validity of IPv4 and the TTL field and conditionally applies the rest of the pipeline.

(action_decomp.p4) Decomposes the L3 table into two tables, (1) a forward table that matches on the IPv4 destination and sets the output port, (2) a rewrite table that matches on the IPv4 destination and performs MAC rewriting.

(metadata.p4) Instead of setting the output port, the L2 and L3 tables set a metadata field. This metadata field is mapped to the output port in the nexthop table, which is applied between the L2 and L3 tables.

(double.p4) Applies all three tables in the pipeline twice.

(choice.p4) Introduces a staging table that sets a metadata variable to select between copies of the abstract pipeline.

We used Avenir to translate 1,001 logical.p4 insertions (1 into PUNT for TTL checking, 500 into L2 for Ethernet destination forwarding, and 500 into L3 for IPv4 destination forwarding and MAC rewriting). We show completion graphs for each target in Figure 6.14.

There are a few things to notice. Every line has an "elbow" at the 50% mark on the y-axis, after which the slope decreases. This represents the transition between parts of the workload. The L3 insertions are slower, because the L2 table is already populated with 500 rules, and slicing has to deal with larger tables. Further, these rules may cause the query template cache to miss: the second "elbow" on the metadata line indicates where the query cache's synthesis engine was able to successfully abstract.

To further demonstrate the power of our template caches, we compare our "cold-start" synthesis (solid lines), where the caches are empty, with "hot-start" synthesis (dotted lines), where the caches are fully populated. We achieve this by running Avenir on the same data twice, without resetting the caches, and logging performance for the second run. The massive performance increase is seen in Figure 6.14. Network operators concerned with nondeterministic runtimes associated with synthesis can manually populate their caches.

6.6.2 Network Emulation

We use Avenir to program the entries of a programmable software switch (bmv2) running in a network emulator (mininet). We configure 64 hosts in a star topology connected to a single switch, and install rules to establish all-pairs ping connectivity. The P4 program running on the software switch is the simple_router.p4 program from the Bmv2 repository. The abstract program is a modified version that joins together the L3 rewriting and forwarding tables into one.

We generate rules required to establish all-pairs connectivity into the logical program and use Avenir to synthesize the equivalent edits into simple_router.p4. We then report the time of the first successful ping between each pair of hosts. We compare Avenir cold-cache run with a manually generated sequence of rule insertions and a pre-populated hot-cache, the results are depicted in Figure 6.15.

6.6.3 Case Study: Trellis & ONOS

Trellis[115] is a set of production-grade SDN apps running on ONOS[17] to provide control plane logic for multi-purpose L2/L3 leaf-spine fabrics of OF-DPA Broadcom switches. Internally, Trellis uses an ONOS API called FlowObjective, designed to allow portability of apps across different switches by abstracting common L2/L3 functionalities. Trellis controls switches by writing FlowObjectives, which are translated by an ONOS driver into OpenFlow messages for OF-DPA tables. Finally, OF-DPA translates OpenFlow messages to Broadcom SDK calls to populate ASIC-specific tables.

We evaluated Avenir on real-world P4 programs that represent the outermost layers of the architecture described above. The fabric.p4 [42] P4 program was created by the ONOS developers to support Trellis on programmable switches. It is designed to simplify control plane operations, and for this reason it closely resembles the FlowObjective API. Likewise, bcm.p4 [84] abstracts tables from the Broadcom SDK, and was created for Stratum [109], an open source switch agent that uses P4 to model control APIs.

We then collected 40k IPv6 route insertions into fabric.p4 corresponding to a switch reboot load test designed by ONOS engineers. Avenir synthesized insertions into bcm.p4 that equivalently process the IPv6 header and egress specification.

Since Avenir does not process the parser, we simulated its behavior by manually setting the validity bit of the IPv6 header to true, and the IPv4 and MPLS headers to false. Further, the P4 specification [28] leaves the initial values of metadata headers undefined; we manually zero-initialize the metadata fields (a behavior that can be specified for many P4 targets via a compile time flag).

Further, we modified the l3_fwd in bcm.p4 by swapping the IPv6 matches for IPv4 matches; otherwise there wouldn't have been a valid translation. Finally since Avenir works with parsed headers, we systematically renamed headers in bcm.p4 to match fabric.p4.

The results are shown in Figure 6.16. According to its engineers, ONOS computes and installs these 40k IPv6 routes over a period of about 15 minutes. This figure includes Trellis' route computation logic, the translation itself and the installation of rules onto the physical target devices. Figure 6.16 shows that Avenir translates these 40k routes into bcm.p4 pipeline in just under 12 minutes. However, it is unclear what conclusions to draw about overhead, because we don't know how

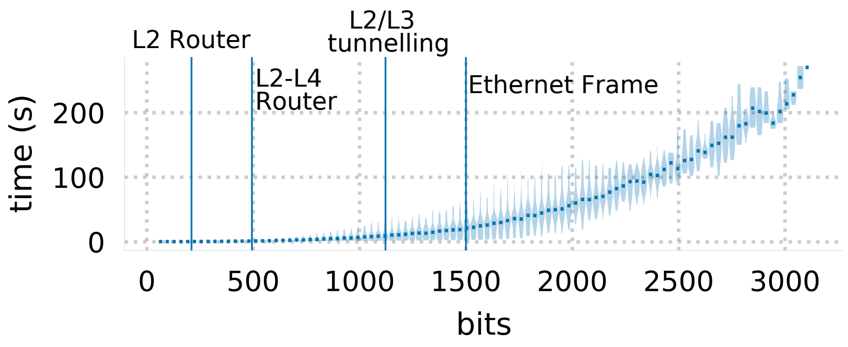


Figure 6.17: Program bits vs time to translate 100 edits. The vertical lines estimate the sizes of common router programs.

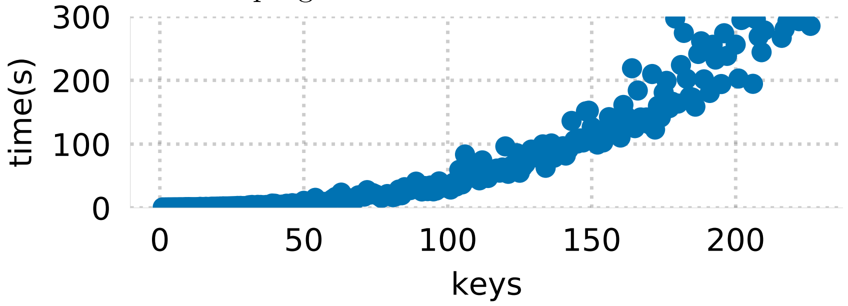


Figure 6.18: Classifier Scaling. We fixed the number of 32-bit output variables to 8, and varied the number of keys.

ONOS' translation logic performs. In the (unlikely) best case, we would have no overhead. In the (also unlikely) worst case, we would nearly double the runtime. The real performance would likely be somewhere between these extremes.

6.6.4 Microbenchmarks

To assess Avenir's scalability, we procedurally generated a collection of microbenchmarks that explore two independent variables, the number of 32-bit input variables I and the number of 32-bit output variables O. For simplicity, the input and output variables sets are distinct.

The abstract pipeline has one table that matches on all of the input variables, and assigns one of the output variables. The target pipeline first matches on all output variables and assigns a metadata value m. This initial staging table is followed by a sequence of O output tables. Table i in this sequence matches solely on m and optionally assigns an output variable.

The results are shown in Figure 6.17. The x-axis shows the number of bits in the abstract program (i.e., $32(I + O)$) and the y axis shows the time in seconds to translate 100 random abstract edits. The violins show the timing distribution marked with median value. The variation comes from the random generation and from the variation in I and O.

Since networking programs are usually classifier-heavy, we also fixed the number of 32-bit output variables to 8, and varied the size of the classifier. The results are in Figure 6.18.

Of course, it's difficult to make general claims about the scalability of Avenir's approach, which incorporates numerous heuristics. Nevertheless, it does seem that the complexity increases exponentially with the number of bits, as is expected for a tool that relies on a black-box solver. Target pipelines with different structure than the regular, repeated structure in our microbenchmarks may behave differently.

6.7 Limitations and Future Work

We discuss two limitations to Avenir's methodology: the cost of formally specificying the abstract and target pipelines, and the run-time overheads of our heuristic search.

The biggest threat to Avenir's use is the requirement that pipelines be formally specified. The work required to develop a formal specification can be significant, and there is no guarantee that a given specification of a pipeline will accurately describe its run-time behavior. Of course, these concerns can be side-stepped if the pipelines are already programmed in P4. But more generally we would need tools for generating specifications and testing conformance. We plan to explore such tools in future work.

Another limitation is our use of heuristic search. The evaluation shows many situations in which Avenir works efficiently, but there are also situations in which it fails to terminate in a reasonable time. For example, to translate from Pipe_1 to OBT in Figure 6.2, Avenir maintains a cross product of L2_fwd and L3_fwd, which requires quadratic operations, and causes incremental heursitics to fail. Expanding the effective scope of Avenir's search is future work. We also plan to explore optimal notions of synthesis—e.g., finding the smallest solution.

6.8 Conclusion

This chapter presented Avenir, a tool that automatically synthesizes control plane operations to ensure uniform behavior across a variety of physical data planes. Avenir uses a counterexample guided inductive synthesis algorithm based on a novel application of sketches to data plane programs. Our evaluation demonstrates that Avenir correctly synthesizes control plane operations with modest overheads.

However, Avenir's termination guarantees are potentially weaker than we would like for a real deployment. The completeness results say that whenever an abstract configuration *can* be translated it *will eventually be*. However, this doesn't give us an abstraction guarantee. We want to know that—that *every* way the controller can configure the abstract pipeline *will* cause Avenir to terminate. In the final chapter of tihs dissertation, we will discuss a framework for synchronizing pipeline programs and *proving* their correctness.

CHAPTER 7

RELATIONAL HOARE LENSES

The field of program logics was established over fifty years ago, grounded in foundational insights such as the following:

> *Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be derived from the text of the program itself by means of purely deductive reasoning.*
>
> —C. A. R. Hoare [56]

Indeed, over the years, programming languages researchers have developed a wide variety of program logics, including probabilistic logics [95, 13], parallel and concurrent logics [91, 64], separation logics [85, 65], and even relational logics [15, 81] that support relating the behaviors of multiple programs. However, with some notable exceptions [34], one aspect of Hoare's vision has been underexplored in prior work: a means to specify and enforce assumptions about the environment in which a program operates.

Suppose the relevant environmental information can be captured in a *configuration*. We can then model a program as a (curried) function from configurations S and inputs A to outputs B:[1]

$$f[\cdot] : S \to A \rightharpoonup B$$

We call f an *open program*, and we write $f[s]$ for the (closed) program resulting from partially applying f to a configuration ($s \in S$). Note that although we could

[1]We write $A \rightharpoonup B$ for the set of partial functions from A to B, and $f(x)\downarrow$ (resp. $f(x)\uparrow$) to mean that f is defined (resp. is undefined, or *diverges*), on input x.

mathematically treat the configuration of an open program as just another part of its input, it is convenient to distinguish between configuring the program (which is an infrequent operation) and executing the configured program (which is a frequent operation).

Many real-world systems can be faithfully modeled as open programs. For instance, consider a network device such as a router, switch, or firewall. The behavior of such a device can be represented as a program in a domain-specific language like P4 [20]. The program specifies the packet-processing behavior of the device, and it also supports dynamic reconfiguration—*e.g.*, setting the next hop toward a given destination. It follows that to fully understand the consequences of executing a program on a given packet, we need more than just the program text. We must also know how the program has been configured. Similar issues arise in other settings—*e.g.*, instruction set architectures (ISA) where the behavior of certain instructions depends on the values in control-status registers (CSRs).

Returning to our example, it is often necessary to maintain two "views" of a network device: an abstract view that provides a high-level configuration API for human operators, and a concrete view that provides a lower-level configuration API that corresponds to its implementation in hardware. Ideally, we would be able to modify the configuration at one level, and have the changes be reflected at the other level, ensuring consistent behavior in the configured programs at all times. However, mapping configurations turns out to be a challenging problem because the high-level and low-level APIs are typically not in one-to-one correspondence. For example, it is sometimes possible to change the next hop to a destination using a single high-level API call, but realizing the same effect at the low level often requires multiple API calls. In Chapter 6, we described Avenir, a solution to the network configuration mapping problem using program synthesis. But while Avenir's synthesis algorithm is sound and complete—*i.e.*, it is guaranteed to find a correct mapping if one exists—it relies on heuristics to speed up synthesis, and provides unreliable performance in practice.

This paper presents a different way to solve the configuration mapping problem, using a novel framework for synchronizing and verifying open programs. Our framework, called *relational Hoare lenses* (RHLenses) is motivated by both theoretical and practical considerations. Theoretically, we aim to integrate relational program logics, which provide tools for reasoning about multiple programs simultaneously, with bidirectional lenses, which provide constructs for mapping between different data views. Practically, we seek to design natural syntax and semantics for RHLenses, and build tools that can solve practical configuration mapping problems in an elegant manner.

The design of RHLenses turns out to be more subtle than it might appear. To illustrate, consider a pair of open programs with the same inputs and outputs but different configurations:

$$f[\cdot] : S \to A \rightharpoonup B \qquad \text{and} \qquad g[\cdot] : T \to A \rightharpoonup B$$

Now suppose we wish to map between S and T so that f and g exhibit identical behavior when instantiated with related configurations. The obvious approach would be to define a lens ℓ between S and T, then use the relation induced by ℓ to prove that f and g equivalent in a relational logic. However, this approach has a key limitation—it lacks modularity. Specifically, using a monolithic lens induces a complex relation on configurations, which complicates the proof of equivalence. Our framework, in contrast, offers compositional primitives that allow users to mix local transformations on configurations and local relational proofs in a seamless manner. Our work also extends lens theory by moving beyond standard round-tripping laws and allowing users to specify custom requirements for configurations manipulated by lenses.

The contributions of this work are as follows:

- We introduce relational Hoare lenses (RHLenses), a framework that unifies relational program logics and bidirectional lenses in a single semantic structure.
- We design a compositional syntax for defining RHLenses, with combinator-based typing rules that capture essential correctness properties.
- We develop a prototype implementation of RHLenses in OCaml and Z3, demonstrating its application to configuration mapping in network data planes.

7.1 Overview

Dynamic reconfiguration of network devices is usually achieved by installing or removing entries in *forwarding tables.* Each entry consists of a *match* and an *action*: the match is a predicate (*e.g.*, checking whether the input packet's destination address matches a given prefix) that determines whether the action should be executed, while the action (*e.g.*, setting the output port) is an imperative procedure that somehow transforms the state of the program. A forwarding table is simply a list of entries. To execute a table, the matches are evaluated in order, and the action corresponding to the first matching entry is executed. (If no entry matches, then executing the table is a no-op.)

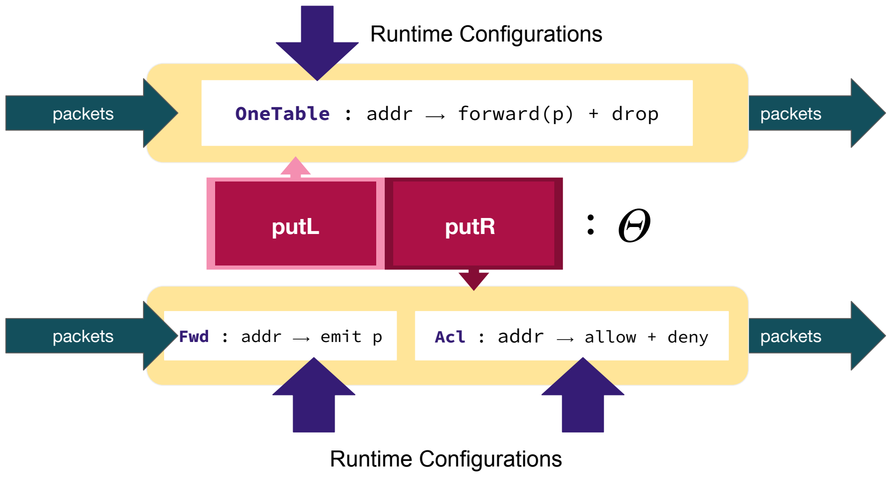


Figure 7.1: A lens synchronizing abstract and concrete views of a packet forwarding pipeline. The relation Θ describes the relation maintained by the lens's *putR* and *putL* functions.

Data planes are often viewed at two levels of abstraction: a high-level abstract view providing a stable interface to human operators and a low-level concrete view describing the software interface[2] to the bare-metal hardware. Consider the example shown in Figure 7.1. In the abstract view, the data plane has a single table `OneTable` that implements forwarding and access control. That is, it matches on the packet's destination address and executes one of two actions: `forward` n, where n is an egress port, or `drop`. In the concrete view, the data plane has two tables, `Fwd` and `Acl` (for *access control list*). As the names suggest, the `Fwd` table implements forwarding, *i.e.*, assigning a forwarding port, while `Acl` implements access control, *i.e.*, dropping unwanted packets. More specifically, `Fwd` matches on the packet's destination address, of type `addr`, and executes a single action, `emit` n, while `Acl` matches on the same value and then executes `allow` or `deny`.

Network operators need to synchronize data between abstract and concrete configurations [2, 17], while ensuring that the configured programs behave the same semantics. So at a high level, we need to relate the abstract and concrete

[2]Sometimes called a switch SDK

configurations as well as the programs that use those configurations.

To relate the configurations, we need to map between abstract and concrete entries in the following way:

OneTable		*Fwd*	*Acl*
$(\texttt{dst}, \texttt{forward}\ n)$	$\leftrightarrows$	$(\texttt{dst}, \texttt{emit}\ n)$	$(\texttt{dst}, \texttt{allow})$
$(\texttt{dst}, \texttt{drop})$	$\leftrightarrows$	$(\texttt{dst}, \texttt{emit}\ \_)$	$(\texttt{dst}, \texttt{deny})$

Note that in mapping the abstract `drop` action to concrete actions, we can use any egress port for `emit`, as the `Acl` table executes `deny`.

To achieve this, we can use the framework of *symmetric lenses* [57], which allow both transformations—*i.e.*, from abstract to concrete configurations, and vice versa—to be specified using a single program. The next section gives a detailed introduction to symmetric lenses. For now, the important thing to understand is that a symmetric lens ℓ describes a pair of functions, $\ell.putL$ and $\ell.putR$, that implement the required transformations. In addition, the semantics of a symmetric lens guarantees that after executing either $\ell.putL$ or $\ell.putR$, the lens will quiesce—*i.e.*, further attempts to propagate changes in either direction will be a no-op. Hence, we can think of a symmetric lens as *maintaining* a relation Θ on configurations—*i.e.*, configurations that become synchronized by propagating changes from one side to the other.

To relate our example data plane programs, we can use techniques from relational logic [15] to relate their behaviors. More formally, we will show that, when instantiated with related configurations, the programs satisfy a relational specification in which the identity relation is both the precondition and the postcondition. That is, the configured data planes process packets in exactly the same way. In our example, we can compute verification conditions for the programs using standard techniques, and then check that they are valid using an SMT solver.

In this example, we separated the tasks of writing a bidirectional lens and reasoning about programs. However, practical examples often do not decompose so neatly. A key challenge is to design a unified framework that allows programmers to implement both tasks using a single abstraction. Achieving this vision in a compositional way is one of the primary goals of this paper.

7.2 Basic Definitions for Relational Hoare Lenses

This section reviews the definition of a symmetric lens and defines what it means for a lens to *maintain* a relation R, before giving a formal definition of relational Hoare lenses.

Definition 7.2.1 ([57]). A *symmetric lens* ℓ from S to T, written $\ell \in S \leftrightarrow T$, consists of a set $\ell.K$ of *complements*, a distinguished element $\ell.k_0 \in \ell.K$, and functions

$$\ell.putL \in S \times \ell.K \to T \times \ell.K$$
$$\ell.putR \in T \times \ell.K \to S \times \ell.K$$

with the following round-trip properties:

1. If $\ell.putL(s, k) = (t, k')$, then $\ell.putR(t, k') = (s, k')$
2. If $\ell.putR(t, k) = (s, k')$, then $\ell.putL(s, k') = (t, k')$

As should be clear from this definition, the *putL* maps data from the source S to the target T, while *putR* maps data in the other direction, from target to source. The complement K stores information that appears only in the source or target, but not both. The symmetric lens laws are somewhat weaker than one might expect, in that they do not stipulate that the information in S must somehow be reflected in T, and vice versa. Rather, the laws only guarantee that after executing a single *putL* or *putR* function, the data in S and T (as well as the complement) will be synchronized, in the sense that invoking *putL* and *putR* again will not change their values.

We will warm up with a few examples of primitive lenses and simple operations on lenses. First, every bijection induces a lens, which does not use its complement—i.e., we may model the complement as the unit value. For example, the *identity lens* is defined as followed:

$$\boxed{\begin{array}{c} \mathsf{id}_S \in S \leftrightarrow S \\ \hline \mathit{putL}(s, ()) \triangleq (s, ()) \\ \mathit{putR}(s, ()) \triangleq (s, ()) \end{array}}$$

The identity lens is so called both because its *putL* and *putR* components are both the identity function, and because it is the identity element for *lens composition*. Given lenses $\ell_1 \in S_0 \leftrightarrow S_1$ and $\ell_2 \in S_1 \leftrightarrow S_2$ we can compose them by taking the product of their complements, and applying their *putL* (resp. *putR*) functions in turn: $\ell_1.\mathit{putL}$ then $\ell_2.\mathit{putL}$ (resp. $\ell_2.\mathit{putR}$ then $\ell_1.\mathit{putR}$). More formally, lens composition is defined as follows:

Given :

$$\ell_1 \in S_0 \leftrightarrow S_1 \quad \ell_2 \in S_1 \leftrightarrow S_2$$

Construct :

$$\ell_1 \circ \ell_2 \in S_0 \leftrightarrow S_2$$

$$\begin{array}{rcl}
K & \triangleq & \ell_1.K \times \ell_2.K \\
k_0 & \triangleq & (\ell_1.k_0, \ell_2.k_0) \\
putL(s_0, (k_1, k_2)) & \triangleq & \textbf{let } (s_1, k_1') \textbf{ be } \ell_1.putL(s_0, k_1) \textbf{ in} \\
 & & \textbf{let } (s_2, k_2') \textbf{ be } \ell_2.putL(s_1, k_2) \textbf{ in} \\
 & & (s_2, (k_1', k_2')) \\
putR(s_2, (k_1, k_2)) & \triangleq & \textbf{let } s_1, k_1' \textbf{ be } \ell_2.putR(s_2, k_1) \textbf{ in} \\
 & & \textbf{let } s_0, k_2' \textbf{ be } \ell_1.putR(s_0, k_2) \textbf{ in} \\
 & & (s_0, (k_1', k_2'))
\end{array}$$

So far, we have not made essential use of the complement. As an example of when the complement is useful, we define a bidirectional projection lens π_t between $S \times T$ and S. To enable the *putR* function to recover the element of T, we squirrel it away in the complement.

Given :

$$t_0 \in T$$

Construct :

$$\pi_{t_0} \in S \times T \leftrightarrow S$$

$$\begin{array}{rcl}
K & \triangleq & T \\
k_0 & \triangleq & t_0 \\
putL((s, t), t') & \triangleq & (s, t) \\
putR(s, t) & \triangleq & ((s, t), t)
\end{array}$$

This gives some motivation for why K is called the *complement*: it stores the data from each side not shared by the other.

We can also compose lenses using the tensor product operator [57]. The resulting lens lets us independently map between different domains. Given two lenses $\ell_i \in S_i \leftrightarrow T_i$,for $i \in \{1,2\}$, the tensor product $\ell_1 \otimes \ell_2$ produces a lens from $S_1 \times S_2$ to $T_1 \times T_2$ by effectively applying the lens components separately to the left and right sides of the pair. The formal definition of the tensor product lens is given below:

Given :

$$\ell_i \in S_i \leftrightarrow T_i, i = 1, 2$$

Construct :

$$\ell_1 \otimes \ell_2 \in S_1 \times S_2 \leftrightarrow T_1 \times T_2$$

$$\begin{array}{rcl}
K & \triangleq & \ell_1.K \times \ell_2.K \\
k_0 & \triangleq & (\ell_1.k_0, \ell_2.k_0) \\
putL((s_1,s_2),(k_1,k_2)) & \triangleq & \textbf{let } t_i, k_i' \textbf{ be } \ell_i.putL(s_i,k_i) \textbf{ in } \textit{for } i = 1,2 \\
 & & ((t_1,t_2),(k_1',k_2')) \\
putR((t_1,t_2),(k_1,k_2)) & \triangleq & \textbf{let } s_i, k_i' \textbf{ be } \ell_i.putR(t_i,k_i) \textbf{ in } \textit{for } i = 1,2 \\
 & & ((s_1,s_2),(k_1',k_2'))
\end{array}$$

7.2.1 Specifications for lenses

As a bidirectional transformation, a lens works to maintain some relation between elements of S and elements of T. (This idea was central to early work on bidirectional transformations[78], which introduced a precursor to lenses under the name *constraint maintainers*.) We formalize this idea as follows.

Definition 7.2.2. Let $\ell : S \to T$ be a symmetric lens. We say that ℓ *maintains* a relation $R \subseteq S \times T$, and we write $\ell \in S \leftrightarrow T : R$, when there exists a set $K_0 \subseteq \ell.K$ satisfying the following:

1. $\ell.k_0 \in K_0$
2. For all $s \in S$ and $k \in K_0$, if $\ell.putL(s,k) = (t,k')$ then $(s,t) \in R$ and $k' \in K_0$
3. For all $t \in T$ and $k \in K_0$, if $\ell.putR(t,k) = (s,k')$ then $(s,t) \in R$ and $k' \in K_0$

By standard properties of inductive definitions, there is a least such relation R, and it is clear that if ℓ maintains R then it maintains any $R' \supseteq R$.

The intuition behind this definition is that if ℓ maintains R, then R should hold of any pair (s,t) that is properly synchronized; that is, any (s,t) such that $\ell.putL(s,k) = (t,k')$ for some k, k', or symmetrically with $\ell.putR$. (The round-tripping laws ensure that the two are equivalent.) The restriction to a suitable set K_0 adds the requirement that in fact k be reachable from $\ell.k_0$ by a sequence of $\ell.putL$'s and $\ell.putR$'s, as well as making the definition invariant under symmetric lens equivalence [57]. Specializing K_0 to all of K, we arrive at the following:

Proposition 7.2.1. *For a lens $\ell \in S \leftrightarrow T$ and elements $s \in S$, $t \in T$, and $k \in \ell.K$, the following are equivalent by the lens laws:*

1. $\ell.putL(s,k) = (t,k)$
2. $\ell.putR(t,k) = (s,k)$
3. *There exists k' such that $\ell.putL(s,k') = (t,k)$*
4. *There exists k' such that $\ell.putR(t,k') = (s,k)$*

Let R_ℓ be the set of all (s,t) such that the above holds for some k. Then ℓ maintains R_ℓ.

The simple description of R_ℓ makes it useful in examples; in particular, we can show that a lens maintains a relation R by showing that $R_\ell \subseteq R$.

Example 7.2.3. The identity lens id_S maintains the diagonal relation, that is, $R_{\mathsf{id}_S} = Id_S$.

Example 7.2.4. The projection lens $\pi_{t_0} \in S \times T \leftrightarrow S$ maintains the relation $\{((s,t),s') \mid s = s'\}$.

Example 7.2.5. Given a lens $\ell \in S \leftrightarrow T : R$ we can define its opposite to be a symmetric lens $(\ell)^{op} \in T \leftrightarrow S$ with the same complement $(\ell)^{op}.K = l.K$, $(\ell)^{op}.k_0 = \ell.k_0$, and swapped put functions: $(\ell)^{op}.putL = \ell.putR$ and $(\ell)^{op}.putR = \ell.putL$. The lens $(\ell)^{op}$ maintains $(R)^{op}$.

Example 7.2.6. Suppose $\ell_1 \in S_1 \leftrightarrow T : R_1$ and $\ell_2 \in T \leftrightarrow S_2 : R_2$. Then $\ell_1 \circ \ell_2$ maintains the composed relation $R_1 \circ R_2 = \{(s_1,s_2) \mid (s_1,t) \in R_1, (t,s_2) \in R_2\}$.

Example 7.2.7. Suppose $\ell_1 \in S_1 \leftrightarrow T_1 : R_1$ and $\ell_2 \in S_2 \leftrightarrow T_2 : R_2$. Then $\ell_1 \otimes \ell_2 \in S_1 \times S_2 \leftrightarrow T_1 \times T_2 : R_1 \Cap R_2$, where $R_1 \Cap R_2 = \{((s_1,s_2),(t_1,t_2)) \mid (s_i,t_i) \in R_i, i = 1,2\}$

7.2.2 Relational Hoare Lenses

Prior work on lenses has mostly relied on general "round-tripping" laws to guide their design [48, 19, 49, 57]. However, when a lens is used with open programs, we can give a more refined characterization of what it means for the lens to be correct—*i.e.*, the programs should satisfy a relational specification whenever their configurations have been properly synchronized using the lens.

Recall the definition of an open program: a curried function $f[\cdot] : S \to A \rightharpoonup B$ which yields a program $f[s]$ of type $A \rightharpoonup B$ upon instantiating it with some configuration $s \in S$. (For concreteness, later, we will use $A = B = \mathsf{Mem}$, denoting programs as stateful, potentially-nonterminating operations $\mathsf{Mem} \rightharpoonup \mathsf{Mem}$). We recall the standard relational Hoare judgment, sometimes called a quadruple:

Definition 7.2.8 ([15])**.** Given $P \subseteq A \times C$ and $Q \subseteq B \times D$, we say that two programs $f : A \rightharpoonup B$ and $g : C \rightharpoonup D$ are *related at* $P \Rightarrow Q$, and write $\vDash f \sim g : P \Rightarrow Q$, if for every $(a, b) \in P$, either both $f(a)$ and $f(b)$ diverge, or $(f(a), g(b)) \in Q$.

Suppose instead $f[\cdot] : S \to A \rightharpoonup B$ and $g[\cdot] : T \to C \rightharpoonup D$ are open programs. Our goal is to synchronize f and g using a symmetric lens ℓ between S and T, while still ensuring that some relations hold on the input and output states. To do this, we will assume that configurations s and t are produced by the lens and check the above semantic judgment on the (now closed) programs $f[s]$ and $g[t]$. We define this formally below:

Definition 7.2.9 (Relational Hoare lens)**.** let $f[\cdot] : S \to A \rightharpoonup B$ and $g[\cdot] : T \to C \rightharpoonup D$ be open programs, and let $\ell \in S \leftrightarrow T : R$ be a lens. We say ℓ *relates* $f[\cdot]$ *to* $g[\cdot]$ *at* $P \Rightarrow Q$, and write $\ell \in f[\cdot] \leftrightharpoons g[\cdot] : P \Rightarrow Q$, when the following holds

$$\forall (s, t) \in R.\ \vDash f[s] \sim g[t] : P \Rightarrow Q$$

Equivalently, if $f : S \times A \rightharpoonup B$ and $g : T \times C \rightharpoonup D$ are the uncurried versions of $f[\cdot]$ and $g[\cdot]$, then we can capture the above using the following definition:

$$\frac{\ell \in S \leftrightarrow T : R \qquad \vDash f \sim g : P \Cap R \Rightarrow Q}{\ell \in f[\cdot] \leftrightharpoons g[\cdot] : P \Rightarrow Q} \tag{7.1}$$

We find this package of a lens specified by a relational Hoare judgment to be a useful abstraction for reasoning about such programs, and so we give it the name *relational Hoare lens* (RHLens). In many instances, relational Hoare lenses may be used in place of a regular relational hoare logic judgment; we give one example here, and delay a more thorough treatment to Section 7.3.

Example 7.2.10 (Trivial relational Hoare lens)**.** Suppose $f : A \rightharpoonup B$ and $g : C \rightharpoonup D$ are closed programs. We can considering as open programs with unit configurations, that is, $f[\cdot] : \mathit{Unit} \to A \rightharpoonup B$ and $g[\cdot] : \mathit{Unit} \to C \rightharpoonup D$, so that they satisfy the judgment $\vDash f[\cdot] \sim g[\cdot] : P \Rightarrow Q$ iff there exists an RHLens $\ell \in P \leftrightharpoons Q : P \Rightarrow Q$.

In Equation (7.1), we have, as a precondition, a regular relational hoare logic judgment. This may be convenient to do when we want to invoke some client relational verifier. However we can also reason solely using lenses, as follows: Suppose we have $f[\cdot][\cdot] : S_1 \to S_2 \to A \rightharpoonup B$ and $g[\cdot][\cdot] : T_1 \to T_2 \to C \rightharpoonup D$, as well as lenses $\ell_1 \in S_1 \leftrightarrow T_1$ and $\ell_2 \in S_2 \leftrightarrow T_2$. We now have the derived rule

$$\frac{\ell_1 \in S_1 \leftrightarrow T_1 : R \qquad \ell_2 \in f^\sharp[\cdot] \leftrightharpoons g^\sharp[\cdot] : P \Cap R \Rightarrow Q}{\ell_1 \otimes \ell_2 \in f[\cdot][\cdot] \leftrightharpoons g[\cdot][\cdot] : \Phi \Rightarrow \Psi}$$

where $\ell_1 \otimes \ell_2$ is the *tensor product lens* from above, and $h^\sharp[\cdot] : T \to (S \times X) \to Y$ where $h[\cdot][\cdot] : S \to T \to X \to Y$, for $h = f, g$, is defined in the obvious way. In this way, in programs with many components, they may be related by lenses one pair at a time. In the next section, well explore a syntax for open programs $f[\cdot]$ and describe a set of lens combinators that exploit the shared structure of programs to obtain modular specifications.

$$\frac{}{\emptyset \models \{\cdot\}} \qquad \frac{f : \mathsf{Value} \to \mathsf{Value}}{\{F \mapsto f\} \models F} \qquad \frac{\sigma_1 \models S_1 \quad \sigma_2 \models S_2}{\sigma_1 \uplus \sigma_2 \models S_1 \cdot S_2} \qquad \frac{\sigma \models S \quad \sigma \in \Theta}{\sigma \models S \ \mathsf{where}\ \Theta}$$

Figure 7.2: A judgment defining whether a configuration is valid *w.r.t.* a schema

7.3 Relational Program Logics

We base our language for open programs on the standard WHILE language. As WHILE is a first-order programming language, we model configurations in terms of uninterpreted functions. The syntax of WHILE (Figure 7.3) is largely standard, though we parametrize it on a set of values (Value) over which the program runs, a set of binary operators Bin over Value, and a set of comparisons Comp over Value, where Comp includes equality. We presume only that each operation $\oplus \in \mathsf{Bin}$ has a total interpretation $\widehat{\oplus} : \mathsf{Value} \to \mathsf{Value} \to \mathsf{Value}$ and each comparison $\sim \in \mathsf{Comp}$ has a total interpretation $\widehat{\sim} : \mathsf{Value} \to \mathsf{Value} \to 2$, where $2 = \{\mathsf{tt}, \mathsf{ff}\}$. From these we construct a language of expressions comprising literal values ($v \in \mathsf{Value}$), variables ($x \in \mathsf{Var}$), binary operators $e \oplus e$ for $\oplus \in \mathsf{Bin}$ and, most importantly, function application $F(e)$.

Function application is precisely where the rubber meets the road. That is, functions define the degrees of "openness" that a program has. For a program c, we can describe its *configuration schema* S via the set of functions $F_1, \ldots, F_n$ that occur in c. We write $\{\cdot\}$ for the empty schema, F for the singleton schema, and $S_1 \cdot S_2$ for the disjoint union of S_1 and S_2, when $\mathrm{dom}(S_1) \cap \mathrm{dom}(S_2) = \emptyset$. Finally, write S where Θ for the refinement of S by Θ, which models constraints on configurations. We write $\mathrm{dom}(S)$ for the set of function symbols that occur in S.

A configuration $\sigma \in \mathsf{Config}$ is a partial function from function symbols to functional values: that is, $\mathsf{Config} = \mathsf{Fun} \to \mathsf{Value} \rightharpoonup \mathsf{Value}$, so formally, $\Theta \subseteq \mathsf{Config}$. To give semantics to schema refinement we need to define what it means for σ to satisfy a schema. We say σ *is a valid configuration of a schema* S, written $\sigma \models S$, if, informally, it defines all the function symbols F in S, and their definitions satisfy all the refinements Θ specified by S. Validity $\sigma \models S$ is defined formally in Figure 7.2.

The value of an expression depends on a configuration and a *memory*, an interpretation of variables $\mu \in \mathsf{Mem} = \mathsf{Var} \to \mathsf{Value}$. The denotation of an expression $[\![e]\!] : \mathsf{Config} \to \mathsf{Mem} \rightharpoonup \mathsf{Value}$ is defined in Figure 7.3: literals v denote themselves; variables x denote lookups $\mu(x)$ in the provided memory μ; operations $e \oplus e'$ recursively evaluate e and e' to values v and v' (if such values exist) and run the interpretation of the operator $v \mathbin{\widehat{\oplus}} v'$; and finally, function application $F(e)$ recursively evaluates e to v, looks up F in the configuration σ to get a function $f : \mathsf{Value} \to \mathsf{Value}$, and runs $f(v)$. The only way the evaluation of an expression can be undefined is if F is not in $\mathrm{dom}(\sigma)$.

Next, we define a minimal syntax of boolean formulae: falsehood $\bot$, implication $\varphi_1 \Rightarrow \varphi_2$, equality of expressions $e_1 = e_2$, and any of the other abstract binary comparators $e_1 \sim e_2$ for $\sim \in \mathsf{Comp}$. We also define the standard derived operations using standard syntactic sugar (*e.g.*, $\neg\varphi \triangleq \varphi \Rightarrow \bot$, $\varphi_1 \vee \varphi_2 \triangleq (\neg\varphi_1) \Rightarrow \varphi_2$, *etc.*). The semantics (defined in Figure 7.3) are standard: $\bot$ denotes ff; $\varphi_1 \Rightarrow \varphi_2$ is tt if φ_2 evaluates to tt or φ_2 to ff; and comparisons $\sim \in \mathsf{Comp}$ are given their semantics by $\widehat{\sim} : \mathsf{Value} \to \mathsf{Value} \to 2$. We assume $= \in \mathsf{Comp}$, with $\widehat{=}$ as Value's equality relation.

A program $c \in \textsc{While}$ can be one of the following: an assignment $(x := e)$, a sequential composition $(c_1; c_2)$, a conditional $(\texttt{if}(\varphi)\{c_1\}\{c_2\})$, or a while loop $(\texttt{while}(\varphi)\{c_0\})$. We write $c\langle S\rangle$ to indicate that c *adheres to* a schema S, which means that the set of function symbols that occur in c is a subset of $\text{dom}(S)$. A program is *closed* if it uses no functions, that is, it adheres to the empty schema $\{\cdot\}$. We will generally use the metavariables S and T to refer to both a schema and to the set of its valid configurations.

The full semantics of programs is given in Figure 7.3. A program c, together with a configuration σ, denotes a partial function on memories $[\![c]\!]^\sigma : \mathsf{Mem} \rightharpoonup \mathsf{Mem}$.

Each component is defined in a standard way, taking the convention that if one required subcomponent is undefined, then the program itself is undefined. For instance, to evaluate an assignment $x := e$ *w.r.t.* $\sigma \in \mathsf{Config}$ and $\mu \in \mathsf{Mem}$, if there exists $v = [\![e]\!]^\sigma\mu$, then we update μ's value of x to be v via the notation $\mu\{x \mapsto v\}$. However, for brevity's sake, we'll simply write $\mu\{x \mapsto [\![e]\!]^\sigma\mu\}$. Briefly, $\texttt{skip}$ denotes the identity function, sequential composition $c_1; c_2$ denotes function composition, the conditional expression $\texttt{if}(\varphi)\{c_1\}\{c_2\}$ evaluates c_1 if φ evaluates to $\mathtt{tt}$, and c_2 otherwise, and finally, $\texttt{while}(\varphi)\{c\}$ denotes a particular least fixpoint, which is computed *w.r.t.* the definedness partial order on $\mathsf{Mem} \rightharpoonup \mathsf{Mem}$.

7.3.1 Relational Hoare Logic

To reason about pairs of programs, the standard approach is to use relational hoare logic, sometimes also called Benton Logic [15]. The principle is this: given two (closed) programs c_1 and c_2, a pre-condition relation P, and a postcondition relation Q, we want to answer the following question: given a pair of state satisfying the input relations, $(\mu_1, \mu_2) \in P$, does running both programs on their respective

Expressions :

$$\begin{array}{rcl} e & := & v \\ & \mid & x \\ & \mid & F(e) \\ & \mid & e \oplus e \end{array}$$

Formulae :

$$\begin{array}{rcl} \varphi & := & \bot \\ & \mid & \varphi \Rightarrow \varphi \\ & \mid & e \sim e \end{array}$$

Commands :

$$\begin{array}{rcl} c & := & \texttt{skip} \\ & \mid & x := e \\ & \mid & c; c \\ & \mid & \texttt{if}(\varphi)\{c\}\{c\} \\ & \mid & \texttt{while}(\varphi)\{c\} \\ & & x \in \mathsf{Var} \quad F \in \mathsf{Fun} \\ S & := & \{\cdot\} \\ & \mid & F \\ & \mid & S \cdot S \\ & \mid & S \text{ where } \Theta \end{array}$$

$\Theta \subseteq \mathsf{Config}$ $x \in \mathsf{Var}$ $F \in \mathsf{Fun}$

$[\![e]\!]^{(-)} : \mathsf{Config} \to \mathsf{Mem} \rightharpoonup \mathsf{Value}$

$$\begin{array}{rcl} [\![v]\!]^{\sigma}\mu & \triangleq & v \\ [\![x]\!]^{\sigma}\mu & \triangleq & \mu(x) \\ [\![F(e)]\!]^{\sigma}\mu & \triangleq & \sigma(F)([\![e]\!]^{\sigma}\mu) \\ [\![e_1 \oplus e_2]\!]^{\sigma}\mu & \triangleq & [\![e_1]\!]^{\sigma}\mu \mathbin{\widehat{\oplus}} [\![e_2]\!]^{\sigma}\mu \end{array}$$

$[\![\varphi]\!]^{(-)} : \mathsf{Config} \to \mathsf{Mem} \rightharpoonup 2$

$$\begin{array}{rcl} [\![\bot]\!]^{\sigma}\mu & \triangleq & \mathsf{ff} \\ [\![\varphi_1 \Rightarrow \varphi_2]\!]^{\sigma}\mu & \triangleq & \begin{cases} \mathsf{tt}, & [\![\varphi_1]\!]^{\sigma}\mu = \mathsf{ff} \\ \mathsf{tt}, & [\![\varphi_2]\!]^{\sigma}\mu = \mathsf{tt} \\ \mathsf{ff}, & \text{otherwise} \end{cases} \\ [\![e_1 \oplus e_2]\!]^{\sigma}\mu & \triangleq & [\![e_1]\!]^{\sigma}\mu \mathbin{\widehat{\sim}} [\![e_2]\!]^{\sigma}\mu \end{array}$$

$[\![c]\!]^{(-)} : \mathsf{Config} \to \mathsf{Mem} \rightharpoonup \mathsf{Mem}$

$$\begin{array}{rcl} [\![\texttt{skip}]\!]^{\sigma} & \triangleq & \lambda\mu.\, \mu \\ [\![x := e]\!]^{\sigma} & \triangleq & \lambda\mu.\, \mu\{x \mapsto [\![e]\!]^{\sigma}\mu\} \\ [\![c_1; c_2]\!]^{\sigma} & \triangleq & [\![c_2]\!]^{\sigma} \circ [\![c_1]\!]^{\sigma} \\ [\![\texttt{if}(\varphi)\{c_1\}\{c_2\}]\!]^{\sigma} & \triangleq & \lambda\mu. \begin{cases} [\![c_1]\!]^{\sigma}\mu, & [\![\varphi]\!]^{\sigma}\mu = \mathsf{tt} \\ [\![c_2]\!]^{\sigma}\mu, & [\![\varphi]\!]^{\sigma}\mu \neq \mathsf{tt} \end{cases} \\ [\![\texttt{while}(\varphi)\{c\}]\!]^{\sigma} & \triangleq & \mathrm{fix}\, \lambda f.\, \lambda\mu. \begin{cases} f([\![c]\!]^{\sigma}\mu), & [\![\varphi]\!]^{\sigma}\mu = \mathsf{tt} \\ \mu, & [\![\varphi]\!]^{\sigma}\mu \neq \mathsf{tt} \end{cases} \end{array}$$

where fix is the least fixpoint operator

$v \in \mathsf{Value}$ $\oplus \in \mathsf{Bin}$ $\sim \in \mathsf{Comp}$

$\widehat{\oplus}$: $\mathsf{Value} \to \mathsf{Value} \to \mathsf{Value}$

$\widehat{(\sim)}$: $\mathsf{Value} \to \mathsf{Value} \to \mathsf{Value}$

$\mathsf{Mem} = \mathsf{Var} \to \mathsf{Value}$

$\mathsf{Config} = \mathsf{Fun} \rightharpoonup \mathsf{Value} \to \mathsf{Value}$

Figure 7.3: Syntax (left), denotational semantics (right) , and auxiliary sets (bottom) for WHILE

inputs ($[\![c_i]\!]\mu_i = \mu'_i$) for $i = 1, 2$ satisfy the output relation, that is, $(\mu'_1, \mu'_2) \in Q$. Notice that this means that we only consider input states on which both c_1 and c_2 converge. We will define a logical judgment $\vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi$ where c_1 and c_2 are (closed) programs, and Φ and Ψ are formulae that denote relations on the inputs and outputs of the programs, respectively.

$$E := v \mid x^{\langle 1\rangle} \mid x^{\langle 2\rangle} \mid E \oplus E \qquad \Phi := \bot \mid \Phi \Rightarrow \Phi \mid E \sim E$$

$$\begin{array}{rcl}
(\!|v|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & v \\
(\!|x^{\langle 1\rangle}|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & \mu_1(x) \\
(\!|x^{\langle 2\rangle}|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & \mu_2(x) \\
(\!|E_1 \oplus E_2|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & \\
\multicolumn{3}{r}{(\!|E_1|\!)^{\sigma}\mu_1\ \mu_2 \mathbin{\widehat{\oplus}} (\!|E_2|\!)^{\sigma}\mu_1\ \mu_2}
\end{array}
\qquad
\begin{array}{rcl}
(\!|\bot|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & \mathsf{ff} \\
(\!|E_1 \sim E_2|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & (\!|E_1|\!)^{\sigma}\mu_1\ \mu_2 \mathbin{\widehat{\sim}} (\!|E_2|\!)^{\sigma}\mu_1\ \mu_2 \\
(\!|\Phi \Rightarrow \Psi|\!)^{\sigma}\mu_1\ \mu_2 & \triangleq & \begin{cases} \mathsf{tt}, & (\!|\Phi|\!)^{\sigma}\mu_1\ \mu_2 = \mathsf{ff} \\ \mathsf{tt}, & (\!|\Psi|\!)^{\sigma}\mu_1\ \mu_2 = \mathsf{tt} \\ \mathsf{ff}, & \mathit{otherwise} \end{cases}
\end{array}$$

Figure 7.4: The syntax (left) and semantics of relational expressions (middle) and formulae (right)

Relational Formulae

We specify input and output relations as quantifier-free first-order formulae over the program variables of the two programs, denoting a function from pairs of memories $(\mu_1, \mu_2) \in \mathsf{Mem} \times \mathsf{Mem}$ to a truth value $\{\mathsf{tt}, \mathsf{ff}\} = 2$. The syntax and semantics can be seen in Figure 7.4. As the two programs may have variable names in common, we annotate each variable used in a formula with which program it comes from, writing $x^{\langle 1\rangle}$ or $x^{\langle 2\rangle}$, *e.g.*. Semantically, this corresponds to $\mu_1(x)$ and $\mu_2(x)$, respectively.

As a convenience, we will define injections from predicates over a single program's variables to relational formulae, which label each variable with a 'side'. The types of these injections are shown below; we omit their (unsurprising) definitions:

$$(-)^{\langle 1\rangle} : \mathsf{Form} \to \mathsf{RelForm} \qquad\qquad (-)^{\langle 2\rangle} : \mathsf{Form} \to \mathsf{RelForm}$$

where Form is the set of formulae φ from Figure 7.3, and $\mathsf{RelForm}$ is the set of formulae Φ from Figure 7.4.

Relational Program Logic

Most of Benton's rules (Figure 7.5) read like standard extensions of the familar Hoare logic axioms. The RHL-Skip axiom says that if both programs are `skip` then the program states don't change. The RHL-Asn axiom runs parallel assignments by subsituting each variable and expression into the appropriate "side" of the relational postcondition Ψ. RHL-Seq sequences c_1 (resp. c_1') with c_2 (resp. c_2') when there exists some intermediate relation that serves as the postcondition of c_1 and c_1' and the precondition of c_2 and c_2'. But the control structures require careful synchronization.

Benton's original axioms and inference rules give us a methodology for analyzing pairs of programs together by exploiting parallel structure between the programs. For instance, RHL-If, reproduced below, relates the "then" and the "else" control flow branches, and uses these relationships to prove the relationship between the full conditionals.

$$\frac{\vdash \Phi \Rightarrow \left(\varphi^{\langle 1\rangle} \leftrightarrow \psi^{\langle 2\rangle}\right) \qquad \begin{array}{l} \vdash c_1 \sim c_1' : \left(\Phi \wedge \varphi^{\langle 1\rangle} \wedge \psi^{\langle 2\rangle}\right) \Rightarrow \Psi \\ \vdash c_2 \sim c_2' : \left(\Phi \wedge \neg\varphi^{\langle 1\rangle} \wedge \neg\psi^{\langle 2\rangle}\right) \Rightarrow \Psi \end{array}}{\vdash \texttt{if}(\varphi)\{c_1\}\{c_2\} \sim \texttt{if}(\psi)\{c_1'\}\{c_2'\} : \Phi \Rightarrow \Psi}$$

This rule gives us a proof that the two programs $\texttt{if}(\varphi)\{c_1\}\{c_2\}$ and $\texttt{if}(\psi)\{c_1'\}\{c_2'\}$ are related at $\Phi \Rightarrow \Psi$ subject to two conditions: first, the conditions must agree, enforcing that the programs branch in the same way, and second, the true (resp. false) branches of each program must be related under the additional assumption that the condition was true (resp. false).

Note that Benton's relational hoare logic inference rules are not complete, and don't aim for any kind of completeness—for example, one could generalize the RHL-If rule to something more complex, writing a rule that compares each pair

$$\frac{}{\vdash x := e \sim y := e' : \Psi[e^{\langle 1\rangle}, e'^{\langle 2\rangle}/x^{\langle 1\rangle}, y^{\langle 2\rangle}] \Rightarrow \Psi}\text{[RHL-}\textsc{Asn}\text{]}$$

$$\frac{}{\vdash \texttt{skip} \sim \texttt{skip} : \Phi \Rightarrow \Phi}\text{[RHL-}\textsc{Skip}\text{]}$$

$$\frac{\vdash c_1 \sim c_1' : \Phi \Rightarrow \Phi' \quad \vdash c_2 \sim c_2' : \Phi' \Rightarrow \Psi}{\vdash c_1; c_2 \sim c_1'; c_2' : \Phi \Rightarrow \Psi}\text{[RHL-}\textsc{Seq}\text{]}$$

$$\frac{\vdash \Phi \Rightarrow \left(\varphi^{\langle 1\rangle} \leftrightarrow \psi^{\langle 2\rangle}\right) \quad \begin{array}{l}\vdash c_1 \sim c_1' : \Phi \wedge \varphi^{\langle 1\rangle} \wedge \psi^{\langle 2\rangle} \Rightarrow \Psi \\ \vdash c_2 \sim c_2' : \Phi \wedge \neg\varphi^{\langle 1\rangle} \wedge \neg\psi^{\langle 2\rangle} \Rightarrow \Psi\end{array}}{\vdash \texttt{if}(\varphi)\{c_1\}\{c_2\} \sim \texttt{if}(\psi)\{c_1'\}\{c_2'\} : \Phi \Rightarrow \Psi}\text{[RHL-}\textsc{If}\text{]}$$

$$\frac{\vdash c \sim c' : \Phi \wedge \varphi^{\langle 1\rangle} \Rightarrow \Phi \quad \vdash \Phi \Rightarrow \left(\varphi^{\langle 1\rangle} \leftrightarrow \psi^{\langle 2\rangle}\right)}{\vdash \texttt{while}(\varphi)\{c\} \sim \texttt{while}(\psi)\{c'\} : \Phi \Rightarrow \Phi \wedge \neg\varphi^{\langle 1\rangle}}\text{[RHL-}\textsc{While}\text{]}$$

$$\frac{\vdash \Phi' \Rightarrow \Phi \quad \vdash c \sim c' : \Phi \Rightarrow \Psi \quad \vdash \Psi \Rightarrow \Psi'}{\vdash c \sim c' : \Phi' \Rightarrow \Psi'}\text{[RHL-}\textsc{Sub}\text{]}$$

$$\frac{\vdash c \sim c' : \Phi \Rightarrow \Psi}{\vdash c' \sim c : \Phi^{op} \Rightarrow \Psi^{op}}\text{[RHL-}\textsc{Sym}\text{]} \qquad \frac{\vdash c \sim c' : \Phi \Rightarrow \Psi \quad \vdash c' \sim c'' : \Phi' \Rightarrow \Psi'}{\vdash c \sim c'' : \Phi \circ \Phi' \Rightarrow \Psi \circ \Psi'}\text{[RHL-}\textsc{Tr}\text{]}$$

Figure 7.5: Structural axioms and inference rules for closed programs [15, 9]

of branches, *e.g.*, including c_1 and c_2'. However, unlike for regular Hoare logic, a completeness result for relational hoare logic is not readily achievable, and so we merely provide a simple yet useful set of sound rules.

7.3.2 Reasoning about Programs with RHLenses

To synchronize open programs, we need to not only declare relationship between our data, but reason about the code that transforms it. We'll define RHLenses and RHLens combinators that mirror the axioms and inference rules from relational hoare logic. To do this, we have concretized the space of our open programs $f[\cdot] : S \to A \rightharpoonup B$ to the semantics of WHILE programs. To synchronize open programs c_1 and c_2 that adhere to schemas S and T respectively, we will write RHLenses ℓ

of the form $\ell \in [\![c_1]\!] \leftrightharpoons [\![c_2]\!] : (\!|\Phi|\!)^{\emptyset} \Rightarrow (\!|\Psi|\!)^{\emptyset}$, where $[\![c_1]\!] : S \to \mathsf{Mem} \rightharpoonup \mathsf{Mem}$ and $[\![c_2]\!] : T \to \mathsf{Mem} \rightharpoonup \mathsf{Mem}$. However, to ease the notational burden, we'll simply write $\ell \in c_1\langle S\rangle \leftrightharpoons c_2\langle T\rangle : \Phi \Rightarrow \Psi$.

To start, we'll specialize some of the lenses from Section 7.2.1 to operate on schemas and their corresponding configurations. Starting simply, we can define a simple identity lens $\mathsf{id}_{c\langle S\rangle}$ that produces a lens witnessing equality of $c\langle S\rangle$. The underlying lens is simply id_S, and the type witnesses extensional equality of c with itself, assuming the equivalence of all variables read (written $\mathsf{reads}(c)$), and demonstrating the equality of all variables written to ($\mathsf{writes}(c)$). We construct this lens as follows. Below, on the left we write the construction of the RHLens $\mathsf{id}_{c\langle S\rangle}$ in terms of the symmetric lens id_S, and on the right we write an inference rule representing the typing constraints on the construction of $\mathsf{id}_{c\langle S\rangle}$.

$$\boxed{\mathsf{id}_{c\langle S\rangle} \triangleq \mathsf{id}_S} \qquad \frac{X = \mathsf{reads}(c) \qquad Y = \mathsf{writes}(c)}{\mathsf{id}_{c\langle S\rangle} \in c\langle S\rangle \leftrightharpoons c\langle S\rangle : \bigwedge_{x\in X} x^{\langle 1\rangle} = x^{\langle 2\rangle} \Rightarrow \bigwedge_{y \in Y} y^{\langle 1\rangle} = y^{\langle 2\rangle}}$$

Additionally, we can write a frame lens that lets us add disjoint specifications. For a set of variables X, we write $X^{\langle 1\rangle}$ or $X^{\langle 2\rangle}$ to indicate that set of variables annotated with a relational "side." Here we overload $\mathsf{vars}(-)$ to compute the variables that occur in programs or free in formulae.

$$\boxed{\|\ell\|^{\Phi} \triangleq \ell} \qquad \frac{\begin{array}{c}\ell \in c\langle S\rangle \leftrightharpoons c'\langle T\rangle : \Psi \Rightarrow \Psi' \\ X = \mathsf{vars}(c) \quad X' = \mathsf{vars}(c') \quad Y = \mathsf{vars}(\Phi) \quad Z = \mathsf{vars}(\Psi) \\ \mathsf{vars}(\Phi') \cap (X \cup X' \cup Y \cup Z) = \emptyset\end{array}}{\ell \in c\langle S\rangle \leftrightharpoons c'\langle T\rangle : \Psi \wedge \Phi \Rightarrow \Psi' \wedge \Phi}$$

Combining this lens with `skip` lets us replicate RHL-Skip via the lens $\|\mathsf{id}_{\texttt{skip}}\|^{-}$.

We can also define a lens for assignment. Given a lens that relates the configurations used in the two assignments and maintains a relation Θ, we substitute the expressions e and e' in for x and y on their respective "sides" of Ψ and check that the preconditon Φ and Θ imply this substitution. Henceforth we'll use the metavariable Θ to refer to symbolic versions of a lens's maintained relation R. We define the underlying symmetric lens below to the left, and to its right, describe the typing constraints that must hold in order to construct lens $\mathsf{assign}\ \ell$

$$\boxed{\mathsf{assign}\ \ell \triangleq \ell} \qquad \frac{\ell \in S \leftrightarrow T : \Theta \qquad \vdash \Theta \wedge \Phi \Rightarrow \Psi[e^{\langle 1\rangle}, e'^{\langle 2\rangle}/x^{\langle 1\rangle}, y^{\langle 2\rangle}]}{\ell \in x := e\langle S\rangle \leftrightarrows y := e'\langle T\rangle : \Phi \Rightarrow \Psi}$$

This is more complicated than the standard relational hoare logic rule, which simply has $\Psi' \triangleq \Psi[e^{\langle 1\rangle}, e'^{\langle 2\rangle}/x^{\langle 1\rangle}, y^{\langle 2\rangle}]$ as the precondition. Morally, this lens composes the synchronization code in ℓ with the assignments, and so we must check that these two pieces compose. This is guaranteed by the assumption that $\Theta \wedge \Phi \Rightarrow \Psi'$.

Next, we will reason about sequential products using RHLenses. The sequential product lens combinator $\ell_1 \rhd \ell_2$ takes a lens ℓ_1 that relates two programs $c_1\langle S_1\rangle$ and $c_2\langle S_2\rangle$ at $\Phi_1 \Rightarrow \Psi_1$ and another lens ℓ_2 that relates $c'_1\langle T_1\rangle$ and $c'_2\langle T_2\rangle$ at $\Phi_1 \Rightarrow \Psi_2$, and uses tensor product[3] to relate the sequential composition of the two programs. Below on the left we can see that the underlying symmetric lens is just the tensor product, and on the right, we show the additional typing constraints required to typecheck that $\ell_1 \rhd \ell_2$ is well-formed—namely that ℓ_1's output relation implies ℓ_2's input relation, and the schemas are disjoint:

$$\boxed{\ell_1 \rhd \ell_2 \triangleq \ell_1 \otimes \ell_2} \qquad \frac{\begin{array}{l} \ell_1 \in c_1\langle S_1\rangle \leftrightarrows c'_1\langle T_1\rangle : \Phi_1 \Rightarrow \Psi_1 \\ \ell_2 \in c_2\langle S_2\rangle \leftrightarrows c'_2\langle T_2\rangle : \Phi_2 \Rightarrow \Psi_2 \\ \vdash \Psi_1 \Rightarrow \Phi_2 \quad \begin{array}{l} \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2) = \emptyset \\ \mathrm{dom}(T_1) \cap \mathrm{dom}(T_2) = \emptyset \end{array} \end{array}}{\ell_1 \rhd \ell_2 \in c_1; c_2\langle S_1 \cdot S_2\rangle \leftrightarrows c'_1; c'_2\langle T_1 \cdot T_2\rangle : \Phi_1 \Rightarrow \Psi_2}$$

[3]Here, for notational brevity, we're abusing an isomorphism between pairs of disjiont configs (σ_1, σ_2) and disjoint unions of configs $\sigma_1 \uplus \sigma_2$

By design, $\ell_1 \rhd \ell_2$ only works with disjoint schemas. This is by design. If we were to allow (*e.g.*) S_1 and S_2 to overlap, say on some symbol F, it's not clear how we should construct F's value in the *putR* direction. Should we use the value in the output computed by ℓ_1? by ℓ_2? Instead, we simply force the program schemas to be disjoint.

We can also use the tensor product lens to relate conditional statements. Similar to Benton's rule, we we synchronize the program branches: ℓ_1 relates the true branches when the conditions hold, and ℓ_2 relates the false branches when they are false. This lens and its preconditions are defined formally below:

$$\boxed{\begin{array}{c} \mathsf{if}(\varphi, \psi)\ \ell_1\ \ell_2 \triangleq \\ \ell_1 \otimes \ell_2 \end{array}}$$

$$\frac{\begin{array}{c} \ell_1 \in c_1\langle S_1\rangle \leftrightharpoons c'_1\langle T_1\rangle : \Phi \wedge \varphi^{\langle 1\rangle} \wedge \psi^{\langle 2\rangle} \Rightarrow \Psi \\ \ell_2 \in c_2\langle S_2\rangle \leftrightharpoons c'_2\langle T_2\rangle : \Phi \wedge \neg\varphi^{\langle 1\rangle} \wedge \neg\psi^{\langle 2\rangle} \Rightarrow \Psi \\ \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2) = \emptyset \\ \mathrm{dom}(T_1) \cap \mathrm{dom}(T_2) = \emptyset \\ \begin{array}{c} c_0\langle S_1 \cdot T_1\rangle = \mathtt{if}(\varphi)\{c_1\}\{c'_1\} \\ c'_0\langle S_1 \cdot T_2\rangle = \mathtt{if}(\psi)\{c_2\}\{c'_2\} \end{array} \quad \vdash \Phi \Rightarrow \left(\varphi^{\langle 1\rangle} \leftrightarrow \psi^{\langle 2\rangle}\right) \end{array}}{\mathsf{if}(\varphi, \psi)\ \ell_1\ \ell_2 \in c_0 \leftrightharpoons c'_0 : \Phi \Rightarrow \Psi}$$

As in sequential composition, we assume that the schemas are disjoint.

To round out our syntactic rules, we can write a lens for reasoning about pairs of while loops. As in Benton's original axioms, we relate the two programs with a relational loop invariant Φ. The underlying symmetric lens is is the same as the one used to relate the loop bodies, which asserts that Φ holds both before and after running the loop bodies. Further, it assumes that if we run the loop bodies when the while condiions are both true, then at the end, the loop conditions are either both true or both false. This ensures that both loops execute in lock-step.

RHL-x	Skip	Asn	Seq	If	While	Sub	Sym	Tr
RHLens	$\|\mathsf{id}_{\texttt{skip}}\|^{-}$	$\mathsf{assign}\ -$	$\rhd$	$\mathsf{if}()$	$(-)^{-}$	$\mathsf{subsume}()$	$(-)^{op}$	$\circ$

Figure 7.6: Correpondence between the RHL proof system and RHLens combinators

We can construct the lens formally as follows:

$$\boxed{(\ell)^{\varphi} \triangleq \ell} \qquad \frac{\vdash \Phi \Rightarrow (\varphi^{\langle 1\rangle} \leftrightarrow \varphi^{\langle 2\rangle}) \quad \ell \in c\langle S\rangle \leftrightharpoons c'\langle T\rangle : \Phi \wedge \varphi^{\langle 1\rangle} \Rightarrow \Phi}{(\ell)^{\varphi} \in \texttt{while}(\varphi)\{c\langle S\rangle\} \leftrightharpoons \texttt{while}(\varphi)\{c'\langle T\rangle\} : \Phi \Rightarrow \Phi \wedge \neg\varphi^{\langle 1\rangle}}$$

The final three Hoare logic axioms are RHL-Sym, RHL-Tr, and RHL-Sub, which allow us to make structural deductions about the relational quadruple itself. RHL-Sym corresponds directly to the $(-)^{op}$ lens, and RHL-Tr to lens composition ($\circ$). RHL-Sub permits weakening of the postcondition and strengthening of the precondition. Our subsumption lens operator does the same, leaving the lens itself unchanged; it is defined below:

$$\boxed{\mathsf{subsume}^{\Phi'}_{\Psi'}(\ell) \triangleq \ell} \qquad \frac{\vdash \Phi' \Rightarrow \Phi \quad \ell \in c_1 \leftrightharpoons c_2 : \Phi \Rightarrow \Psi \quad \vdash \Psi \Rightarrow \Psi'}{\mathsf{subsume}^{\Phi'}_{\Psi'}(\ell) \in c_1 \leftrightharpoons c_2 : \Phi' \Rightarrow \Psi'}$$

In Figure 7.6 we summarize the relationship between Benton's logic and the combinators we presented in this section. The correspondence with Benton's relational Hoare logic lets us conclude that RHLenses can be used to reason effectively about large programs. These operations are structured syntactically—*i.e.*, they follow the structure of the programs and allow us to provide proofs about them. In the next section, we'll present some examples of lenses that manipulate the configurations in interesting ways.

7.4 Implementation

```
module Type = sig
  type t
  ...
end
module Var = struct
  type t = {
    name: string;
    t: Type.t
  }
end
module Sym = struct
  type t = {f: string;
            i: Type.t;
            o: Type.t}
end
```

```
module type Expr = sig
  type t
  val bv : int -> width -> t
  val var : Var.t -> t
  val ( $ )  : Sym.t -> Var.t -> t
  val ( $$ ) : Sym.t -> t -> t
  val ( - )  : Expr.t -> Expr.t -> t
end
module type Form = sig
  type t
  val true_ : t
  val ( == ) : Expr.t -> Expr.t -> t
  val check : Var.t -> t
  val ands : Form.t list -> t
end
```

```
module type Imp = sig
  type t
  val skip : t
  val (<~) : Var.t -> Expr.t -> t
  val seq : t list -> t
  val ite : Form.t -> t list -> t list -> t
end
```

Figure 7.7: Summary of the core data structures and smart constructors in Spectacle.

Our implementation, Spectacle, is an OCaml instantiation of RHLens over a simple loop-free imperative language with bitvectors (`Imp`, Figure 7.7). We implemented the core operators from Section 7.3.2 in an LCF-style architecture with an abstract type of RHLenses, and call out to an SMT solver for proofs in the base logic. Importantly, our implementation allows users to provide custom, trusted specified lenses, whose specifications are not verified by Spectacle, but it verifies all subsetquent reasoning, including all RHLens reasoning. In the remainder of this section, we'll use a literate style to desribe Spectacle's core types.

First, we define our core data type for lenses in the `Lens` module. `Lens` comprises a single type definition `('k, 'typ) t`, which is a record that has the three

```
module type Cfg = sig
  type fn = Value.t -> Value.t
  type t = (Sym.t * fn) list
  val set : t -> Sym.t -> fn -> t
  val get : t -> Sym.t -> fn
end
```

```
module type Lens = sig
  type ('k, 'typ) t = {
    typ : 'typ;
    missing : 'k;
    putL : Cfg.t * 'k -> Cfg.t * 'k
    putR : Cfg.t * 'k -> Cfg.t * 'k
  }
end
```

Figure 7.8: Modules describing Lenses (right) and the configurations they act upon (left)

lens components, `missing`, `putL`, and `putR`, as well as an explicit `'typ` field, which we will customize to define our respective typing disciplines for lenses. The definition of `Lens` is shown bon the right side of Figure 7.8.

On the left, in Figure 7.8, is the `Cfg` module, whose type `t` describes configs our lenses synchronize. A `Cfg.t` associates `Sym.t`s with OCaml function on our value type (`Value.t`) that they interpret. The `get` and `set` functions enforce that the config is a well-formed association list (*i.e.*, no duplicate `Sym.t`s) Notice that there are some well-formedness constraints that in this prototype must be checked by hand: for a Sym-valuation pair `f`$, v$ it must be the case that v is a function from values of types `f.ins` to a value of type `f.out`. We also must check the symmetric lens laws by hand.

Now, to describe how lenses maintain relations, we'll define the configuration schema, `Schema`, of an open program. A `Schema.t` describes the function symbols (`symbols`) that can occur in a program, as well as a First-order logic refinement (`refine`) on those symbols.

```
module type Schema = sig
  type t = {symbols: Sym.t list; refine: Form.t}
end
```

The `symbols` field of the record type is a list of `Sym.t`s, each of which indicates a function symbol and its type (c.f. Figure 7.7). The `refine` record is a formula (`Form.t`) that indicates a refinement over those symbols specified in `symbols`.

Now we can supply lenses with logical specifications over their function symbol. Such a spec is represented by the `Spec.t` type below, comprising schemas for each program being related (`left`, and `right`) respectively, and a formula `spec` relating those schemas.

```
module type Spec = sig
  type t = { left: Schema.t; right: Schema.t; spec: Form.t; }
end
```

We use this to concretize the `'typ` field to produce the `'k SpecLens.t` type, which corresponds to defines a lens $\ell \in S \leftrightarrow T : \Theta$. In Spectacle, a lens `l : 'k SpecLens.t`, where `'k` is the type of the complement, `l.typ.left` corresponds to S, `l.typ.right` corresponds to T, and `l.typ.spec` corresonds to Θ. We can see this below:

```
module type SpecLens = sig
  type 'k t
  val get_lens : 'k t -> ('k, Spec.t) Lens.t
  (* PRE: the lens satisfies its spec *)
  val trusted  : ('k, Spec.t) Lens.t -> 'k t
```

To maintain the invariant that `'k SpecLens.t` lenses are correctly specified, we leave the type abstract, but expose a function to allow users to provide their own trusted specified lenses. Now, we can implement a few useful lenses with specifications. For instance, the identity lens id_S described above is given below:

```
  val id : Schema.t -> unit t
```

The tensor product lens combinator $\otimes$ is given below as well.

```
  val ( * ) : 'k1 t -> 'k2 t -> 'k1 * 'k2 t
```

The copy lens is a bijective RHLens that takes two symbols F and G as parameters and maps valid configurations σ of F to a valid configurations of G constructing $\{G \mapsto \sigma(F)\}$. The inverse direction is the same, with F and G reversed.

```
  val copy : Sym.t -> Sym.t  -> unit t
```

The duplication lens `dup` is also a bijective RHLens that takes a symbol F and copies it (as above) into configurations for the schema $G_1 \cdot G_2$ where $G_1 \equiv G_2$, by copying the configuration F into both G_1 and G_2.

```
  val dup : Sym.t -> Sym.t * Sym.t -> unit t
  ...
end
```

Now to reason relationally about programs, we need a more substantive model of open programs. We define a simple loop-free imperative language (`Imp`) over fixed width bitvectors. The types of the core smart constructors can be seen in Figure 7.7.

And now we can write down our relational hoare logic quadruple, which takes two programs (`left` and `right`), a precondition (`pre`) and a postcondition (`post`).

```
module type Benton = sig
  type t = {left: Imp.t; right: Imp.t; pre: Form.t; post: Form.t;}
  val equal : Imp.t -> Imp.t -> t
  ...
end
```

We define the `equal` constructor to help us reason about program equality. It takes two imperative programs and constructs a quadruple that holds if and only if the two programs are equivalent.

Now we can create a relational Hoare lens type `RH` which comprises a specification type (`spec`) and a relational Hoare quadruple (`benton`)

```
module type RH = sig
  type t = { spec : Spec.t; benton : Benton.t }
end
```

Finally, our relational Hoare lenses are created by passing the `RH.t` in as the type argument to `Lens.t`. Keeping `'k t` an abstract type lets us maintain the invariant that the lens actually satisfies its RHLens type.

```
module type RHLens = sig
  type 'k t
  val get_lens : 'k t -> ('k, RH.t) Lens.t
```

Now we can write down the types of some standard operators. As before, we can directly state the identity lens. Notice that we only need to pass the program `Imp.t` because we can compute its schema by reading off the functions it uses.

```
  val id : Imp.t -> unit t
```

We can define composition in largely the same way as $\rhd$. Importantly, we need to check the implication between the relational post-condition of first lens and the relational pre-condition of the second. To do this we generate an SMTLIB expression, and discharge it using the Z3 SMT solver. The type of this operator is shown below:

```
  val ( *> ) : 'k1 t -> 'k2 t -> ('k1 * 'k2) t
```

And we can define our injection operator (`|=`), which performs monolithic reasoning given a `SpecLens` and the `Benton` quadruple that it should satisfy. They type of (`|=`) is shown below:

```
  val ( |= ) : 'k SpecLens.t -> Benton.t -> 'k t
```

To check the safety of an injection `l |= b` we must verify that assuming *l.typ.spec* implies that `b` is a valid relational Hoare quadruple. To perform this check, we construct the product program `b.left`×`b.right`, and compute its weakest precondition with respect to `b.post`. This leaves us with one final obligation: checking that `b.pre` implies the weakest precondtion. To discharge this assumption, we generate an SMTLIB program and use Z3 to check its validity.

```
  val ( |> ) : 'k1 SpecLens.t -> 'k2 SpecLens.t -> ('k1 * 'k2) Speclens.t
```

Finally we provide a specialized kind of frame rule. We provide a list of variables `xs` and produce a formula that establishes the relational equality between the two programs. Then, the frame operator `|&|` lets us conjoin this formula to both the pre- and post- conditions of an RHL `l` as long as those variables `xs` do not occur anywhere in the program in `l`'s type.

```
  val ( |&| ) : 'k RHLens.t -> Var.t list -> 'k RHLens.t
end
```

7.5 Case Study: Network Data Plane Programs

In networking, the separation between the control plane and the data plane requires programmers to relate open programs. In the control plane, high level routing algorithms, compute routing preferences, which then produce dynamically-changing forwarding tables that instruct the data plane how to forward packets. Often, to support a diverse set of switch pipelines, network engineers will write their control logic against a unified interface such as the Open Compute Project's Switch Abstraction Interface (SAI) [87]. Then, engineers will write complex and error-prone driver code to translate between the abstract switch and the concrete one.

As a case study, we will show that we can write a concise `RHLens.t` that maintains (and proves) equivalence for dataplane programs. We draw our examples from Avenir [22], which used program synthesis to map configurations in only one direction. Our mappings must be written by hand, but provide verified bidirectional transformations with predictable performance.

7.5.1 Source Program

The qualitative benchmarks used in Avenir mimic its deployment in a network with a single unified abstration. We call this abstraction program `source`, and we translate its tables to three "target" programs `action_decompose`, `lag_decompose`, and `early_validate`. Each program is composed of three standard internet processing blocks: an Ethernet forwarding block, the an IPv4 routing block, and a validation block. We define each of these in turn.

The Ethernet block has a very simple schema comprised of two configurable functions `eth_act` and `eth_port`, which each take the `48`-bit Ethernet Destination address: `eth_act` computes a bit indicating whether the packet should be forwraded, and `eth_prt` computes the 9-bit forwarding port. The Ethernet block applies these functions to the `eth_dst` variable (definition elided) using the smart constructor `$` specified in Figure 7.7, which constructs an `Expr.t` corresponding to the application of a function symbol to a single variable. This program proceeds as follows: if `eth_act $ eth_dst` is 1, the program assigns `eth_act $ eth_dst` to the `port` variable.

```
let e_sch = Schema.{
  symbols = [eth_act; eth_dst];
  refine = Form.true_
}
```

```
let eth =
  ite (check (eth_act $ eth_dst)) [
    port <~ (eth_prt $ eth_dst)
  ]
  []
```

As a sanity check, we can very easily produce the identity lens. All the checks here are syntactic, which means there's no need to invoke Z3.

```
let eth_id : unit RHLens.t = RHLens.id eth
```

The IP block's schema has three functions: `ip_act`, `ip_port` and `ip_dst`, whose definitions are elided. These three functions read the 32-bit `ipv4_dst` field, then `ip_act` returns a bit that determines whether the packet should be forwarded. The forwarding logic consists of setting the port field to `ip_port $ ipv4_dst`, copying the `eth_dst` field into the `eth_src` variable, and updating the `eth_dst` address with `ip_dst $ ipv4_dst`, and finally decrementing `ipv4_ttl` by 1 (represented as the 8-bit bitvector `(bv 1 8)`). The IP block (`ip`) is below shown to the right:

```
let vld =
 ite (check (valid $ ipv4_ttl))
  [port <~ bv 511 9]
  []

let source = seq [eth; ip; vld]
```

```
let ip = seq [
  ite (check (ip_act $ ipv4_dst)) [
    port <~ ip_port $ ipv4_dst;
    eth_src <~ eth_dst;
    eth_dst <~ ip_dst $ ipv4_dst;
  ] [];
  ipv4_ttl <~ ipv4_ttl - bv 1 8
]
```

Finally the validation block (`vld`, above left) has had a single function `valid`, which reads the IPv4 Time To Live (`ipv4_ttl`) field, and, returns a single bit, which decides whether the packet should be assigned the virtual port value `bv 511 9`, which indicates that the packet should be "dropped," that is, it should not be forwarded.

7.5.2 Action Decompose

Our first target data plane program is called `action_decompose`, it is structurally similar to `source`, except that for its IP processing logic. `action_decompose` decomposes the logic into two blocks: `fwd` and `write`. The `fwd` block is solely responsible for computing the value of `port` from the fucntions `fwd_act` and `fwd_prt`, while the `write` block is responsible for the remainder of the rewrites, and uses the functions `write_act` and `write_dst` to do so:

```
let fwd =
 ite (check (fwd_act $ ipv4_dst))
  [port <~ fwd_prt $ ipv4_dst]
  []
let action_decompose =
  seq [ eth; fwd; write; vld ]
```

```
let write = seq [
 ite (check (wrt_act $ ipv4_dst)) [
   eth_src <~ eth_dst;
   eth_dst <~ wrt_prt $ ipv4_dst;
 ] [];
 ipv4_ttl <~ ipv4_ttl - bv 1 8
]
```

Then, the `action_decompose` pipeline (defined above) is similar to `source`, differening only in that the `ipv4` block has been replaced with `fwd` and `write`.

Now, our goal is to synchronize the functions in `action_decompose` and `source` in a way that makes the programs functionally equivalent. That is, we need a lens that satisfies the following relational Hoare quadruple:

```
let b = Benton.equal source action_decompose
```

For equivalence to hold between the two programs, `write_act`, `ip_act`, and `fwd_act` must be equivalent. We can produce this outcome using the `dup` constructor below, which produces a lens between `ip_act` and `fwd_act` · `write_act` where `fwd_act` ≡ `write_act`.

```
let act_dup = dup ip_act fwd_act write_act
```

Next, we can relate `ip_prt` and `ip_dst` with `fwd_prt` and `ip_dst` using the `copy` lens.

```
let copies = copy ip_prt fwd_prt * copy ip_dst write_prt
```

As a first pass, we could reason about the equivalence of these `source` and `action_decompose` monolithically, by writing the following lens:

```
(id eth_schema * act_dup * copy * id vld_schema)
  |= Benton.equal source action_decompose
```

which would compute the weakest precondition of `source` × `action_decompose`, and use a solver to discharge the induced implication.

But this fails to make use of the fact that we can align the programs with `*>` and (*i.e.*, without invoking a solver) show that the ethernet and validation blocks are equivalent. Then, we'll only need to invoke the solver to show that the specs of `act_dup` and `copies` are sufficient to prove equality of `ip` and `fwd;write`.

We can build our final relational Hoare lens as follows:

```
(id eth_block |&| [eth_dst; eth_src; ipv4_ttl])
*> (act_dup * copies |= Benton.equal ip_block fwd_write_block)
*> (id vld_block  |&| [eth_dst; eth_src; ipv4_ttl])
```

Notice that we used the frame operator `|&|` which conjoins relational equality of the supplied list of variables to the pre and post conditions of the lens, as long as the variables do not occur anywhere in the lens definition.

7.5.3 Metadata Decomposition

First, we're going to rephrase our source IP block using only a single table `ip` that returns the bitvector concatenation of the results of `ip_act`, `ip_prt` `ip_dst`. Note that it's easy to write a lens that witnesses this equality using the fact that $2^n \times 2^m$ and 2^{n+m} are isomorphic. We omit this for brevity.

This single-function IP block is shown below to the right. To its right, we show the decomposition ofthe `ip_dst` table into two tables, a `group` table, that computes a `group` from `ip_dst`, and a `group_fwd` table. This pattern is useful for engineers who want to assign IP addresses to semantic groups (via `group`), and then route those semantic groups together (via `group_fwd`).

```
let decr = ipv4_ttl <~ ipv4_ttl - bv 1 8
let extr = seq [a <~ extract 0 0 data;
                p <~ extract 1 8 data;
                d <~ extract 9 57 data]
let blck = seq [
  ite (check a) [port <~ p;eth_src <~ eth_dst;eth_dst <~ d][];
  decr]
let ip' = seq [data <~ ip $ ip_dst; extr; blck]
let grp = seq [group_id <~ group $ ip_dst;
               data <~ group_fwd $ group_id;
               extr;blck]
```

As before, we'll use `*>` to leverage the fact that the Ethernet and validation blocks are equivalent, and focus our attention on relating `grp`. We'll need to define a custom lens `decompose` that relates a single-function schema $F : X \to Y$ and a two-function schema $G_1 : X \to Z, G_2 : Z \leftrightarrow Y$. Note that we've assumed that G_2 is invertible. This will allow us to soundly recover F. Fundamentally, leftwards direction will compute $F := G_2 \circ G_1$ and store G_2 in the complement. Then the rightwards direction will take G_2 out of the complement and compute $G_1 := G_2^{-1} \circ F$.

In Spectacle, we can ensure specify the invertability of a function `f` by declaring another function `g` and asserting that it is f's inverse. The `inverses` schema generator defines this below:

```
let inverses f g =  Schema.{symbols=[f; g]; refine= ands [
    forall x (var x == f $ g $ var x ); forall y (var y == g $ f $ var y )]}
```

where `x` is a variable of the input type of `g` and `y` of `f`.

The type of our decompse lens is below:

```
val decompose : Sym.t -> Sym.t * Sym.t -> Cfg.fn * Cfg.fn
              -> (Cfg.fn * Cfg.fn) SpecLens.t
```

Note that the lens requires an initial invertible function pair (g, g^{-1}) to work. These are used to populate the lens' `missing` component. We'll simply provide the identity function here–writing `p_id` to indicate this pair. The final lens is shown below

```
(id eth |&| [eth_dst; eth_src; ipv4_ttl])
*> ((decompose ip group lag_fwd lag_fwd_inv p_id)
                        |= Benton.equal ip' lag)
*> (id vld_block |&| [eth_dst; eth_src; ipv4_ttl])
```

Note that we must augment the `SpecLens` with the appropriate proof that the relationship shows that `ip'` and `lag` are equal.

7.5.4 Early Validation

Our final example will require us to reason monolithically, because it requires a full-program transformation. We would like to be able to use our "swap" lens combinator (`>*<`) which, via the underlying tensor product combinator, composes the programs on each side in opposite order. The catch is that it assumes the programs on each side, and that specifications use disjoint variable sets. Unfortunatley, in this example, moving the validation table from the end of the pipeline to the beginning requires interaction with the ttl decrementation logic from the `fwd` block.

Instead we'll observe that in the domain of bitvectors (with wraparound semantics), subtraction and addition are inverses of each other. So, we can push the validate function back through the decremenation by simply decrementing the input. Of course, going the other direction, the input must be incremented prior to being passed into the validate function. Spectacleprovides a lens `incr_dom` that does this: it has the following type:

```
val incr_dom : Sym.t -> unit SpecLens.t
```

We can then use invoke a solver to show that this lens proves the safety of swapping `valid` table and the `ipv4_ttl` decrementation:

```
let validate = drop <~ vld $ ipv4_ttl
let drop_invld = ite (check drop)
  [drop <~ BV(511,8)] []
```

```
let ttlvld =
  incr_dom |= Benton.equal
    (seq [decr; validate])
    (seq [validate; decr))
```

Then we can use the swap lens combinator (`>*<`) to push `validate` up to the beginning of the pipeline since it uses disjoint variables from `eth` and `ip`. The final lens[4] is the following:

```
(id_pipe (seq [eth;ip'']) *> ttlvld *> drop)
|> ((id_pipe (seq [eth; ip'']) >*< id_pipe validate)
    *> id_pipe drop_invld)
```

where `ip''` is the same as `ip`, but with the terminal ttl decrementation command removed. This lens produces a proof of the equivalence of `source` and `seq[validate;eth;ip;drop_invld]`.

[4]ignoring the required framing lenses

CHAPTER 8

RELATED WORK

Here we summarize the key lines of research related to this thesis.

Analysis of Open Systems Much work has been done in verifying and reasoning about open programs, that is programs with some unresolved identifiers, such as programs that make use of libraries, modules, *etc.* In general, treating these functions as true black boxes produces a plethora of false positives. This has been observed in general purpose programming [34], as well as in the domain of network programming [72]. The Saturn project is a general purpose framework for the static analysis of programs that computes program summaries to constrain code with unknowns [119, 34]. Over the years, various techniques have been proposed to compute specifications of increasingly high quality for the unresolved identifiers in programs, whether they be maximal (subject to syntactic constraints) [4], necessary [36], or weakest [23]. The JIST tool leverages bounded model checking in temporal logics to learn automata that describe interface specifications [5].

Relational Verification Tony Hoare's original axioms for program verification provide a robust foundation for reasoning about single programs. Indeed, a series of papers by Barthe and co-authors have investigated the properties and construction of product programs to verify relational properties of pairs of programs using single-program techniques [10, 11].

The original relational Hoare logic axioms were produced by Benton in 2004 [15] as part of a larger paper describing simple proof systems for pairs of programs. These proof systems have been expanded on over the years moving beyond Benton's syntactically driven rules, introducing rules that manipulate a single program at a time [11, 12], or provide general purpose structural properies of program pairs, we a modernized [9] version of Benton's original proof system [15].

Relational verification problems for open systems occur quite often in the domain of databases. In fact our terminology of "schema" is borrowed from this domain. However instead of "configurations" they refer to "instances." There are many techniques for relating two programs over disparate schemas such as data migration [96], data exchange/integration [44, 43], schema mappings [79], as well as techniques for verifing the equivalence of systems over disparate schemas [117]. However, the effectiveness of these techniques relies on domain-specific assumptions about databases such as their finite-relational structure, and robust logical interfaces (*e.g.*, SQL, Datalog). The RHLens framework proposed in this paper is general purpose, allowing robust, terminating data synchronization to maintain relational properties.

Bidirectional Programming Over the past two decades, lenses have become a key abstraction for bidirectional data transformations. Initially introduced to reason about the *view-update problem* in databases, ensuring that changes in one view of the data are reflected in the data source [48]. This foundational work on lenses has been extended in several ways, notably with symmetric lenses, which allow bidirectional updates to be applied consistently in both directions, ensuring round-trip correctness [57]. Our work is based on this core symmetric lens formalism. Another major development is the introduction of quotient lenses, which generalize lenses by allowing transformations that respect equivalence relations on data structures, providing a more flexible mechanism for bidirectional updates in the presence of complex data equivalence [49]. Our work is the first to describe combinators for lenses that preserve relational properties on the programs that invoke the synchronized data.

Synthesis. Avenir is based on Sketching [103], wherein the programmer is allowed to insert unknown "holes" into a program that are filled using CEGIS [104]. Sketching has been used to build a code generator for packet-processing switch pipelines [53]. NetComplete [39] allows network operators to express their intent by sketching parts of the intended configuration for refactoring or updating purposes. Our novelty is to use sketching to synthesize control plane mappings.

Another use of synthesis is to generate implementations from high-level specifications, e.g., stratified Datalog[40], regular expressions with uninterpreted functions [98], first-order logic constraints [18], and LTL [74].

P4 Verification. There are several recent projects on verifying P4 program properties. Lopes et al. developed an operational semantics for P4 and developed a verification tool based on Datalog which can check program equivalence [77]. P4K presented an operational semantics for P4 using the K framework [66]. p4pktgen uses symbolic execution to generate test cases for P4 programs [83]. Our symbolic compilation is informed by previous work [72, 112, 36, 108, 38, **?**], though we are the first to prove our modeling approach correct. The `p4v` paper informally posed the problem of ci-spec inference [72]. The Π4 paper presents a dependent refinement type system for modular verification in the style of `p4v` [38]. The `p4-constraints` library offers a language for specifying ci-specs [107], but the language is semantically restricted and does not provide an inference mechanism. The `p4testgen` tool generates test cases for P4 programs, and can reason about ci-specs expressed in `p4-constraints` to reduce false alarms [97].

Network Virtualization. There are many SDN controllers, such as POX [93], NOX [55], and Open Daylight [89]. A few of them specifically target the problem of flow rule composition, including the Frenetic language and controller [51] and Pyretic [80]. Other efforts have focused on network virtualization, i.e., mapping abstract specifications down to target realizations, such as ONIX [69]. FlowVisor [99], CoVisor [61] and the NetKAT compiler [102]. Among this work, Avenir is unique in developing an approach to managing heterogeneous abstract and target pipelines.

Logical Abduction. QE-driven maximal spec inference has been well-studied in the formal methods literature [3, 35, 33]. In particular, the MaxSafeSpec algorithm can be used to produce "maximal" conjunctions of single-table specs.[1] Here, maximal does not mean "weakest," rather, it means that none of the single-table conjuncts can be safely weakened. This notion is indeed stronger than weakness: "maximal" specs are often non-trivially more restrictive than the weakest specs.

Specification Inference. The spyro tool [92] provides a general-purpose framework for spec synthesis which summarizes arbitrary queries from given DSL. In contrast, while our work is specialized to the domain of data planes, Capisce infers a precise ci-spec, without requiring a specific DSL. Further, while spyro's algorithm uses a syntactic CeGIS algorithm, our approach is more semantic—we compute ci-specs using deductive tools: symbolic analyses and Qe.

Finally, *Config2Spec* [18], infers control properties f traditional networks using a refinement loop that uses both emulation and verification to generate high quality

[1]The MaxSafeSpec algorithm uses general functions as its core model. In our domain we would specialize their general functions to table calls

properties. *Config2Spec* focuses on network-wide control properties, while Capisce focuses on safe configs for individual switches.

Control Plane Verification & Synthesis NetKAT takes a (co-)algebraic approach to specifying, verifying and compiling network-wide control planes [6]. Work on synthesizing consistent updates shows how to synthesize network updates so that each packet views a consistent snapshot of the network [75]. Genesis [110], NetComplete [39], and Propane/AT [14] all synthesize legacy network configs from high-level specifications. Recent work on P4R-Type [71] develops a typed variant of P4Runtime, the generic control-plane API used by P4 programs. While P4Runtime enforces some type constraints dynamically, P4R-Type guarantees that type errors will not arise at runtime.

CHAPTER 9

CONCLUSION

If there is a guiding light of networking design, it is Jon Postel's Robustness Principle:

Be conservative in what you do,
be liberal in what you accept from others.

—Jon Postel

As applied to traditional networking, this maxim covers packet parsing, and packet processing. To be liberal in what you accept from others, in the networking domain, is to accept any packet shape that the network will throw at you, no matter how maliciously designed or how ill-intentioned it is. The point is to have robust packet parsing and inspection logic that will handle *all* packets. The flipside of this maxim is to assume that other network devices have not been so well-designed. Rather than using obscure protocols or type conditions on packets, network devices should endeavor to use the most standard and simple solution to their problems.

As we've seen throughout the course of this dissertation, this maxim does not apply to our *subprime meridian*: the interface between the control and the data plane. Instead, the onus is on the control plane to be conservative in how it configures the data plane. This dissertation has provided a series of techniques that help the control plane do just that.

SafeP4 In Section 2.1 and Chapter 3, we developed an occurence-style type system for data plane programs to catch bugs related to header validity. We characterized a set of reasonable assumptions about the control plane's interactions with the data plane, forming a simple kind of specification inference. We showed that we were able to detect and fix errors in programmable data planes. However, the specifications we computed we simple and heuristic.

Capisce We generalized this simple approach in Chapter 5, using our Capisce tool, which computes the weakest, safe control interface spec (ci-specs). Further, the ci-specs are *efficiently control monitorable*, that is, they have polynomial expression complexity. We showed that we could compute ci-specs for industrial and academic research programs. Now, data plane engineers can clearly comunicate their configuration assumptions to the control plane engineers.

Avenir Not to leave the control engineers high and dry, we described a system for computing configurations that satisfy a given specification, optimizing for practical use cases where that specification is expressed in the form of an abstract packet processing pipeline. In Chapter 6 we developed a tool, Avenir, based on counterexample guided synthesis, which can efficiently compute specifications for a wide variety of targets, including industrial-grade targets, with minimal overhead. However, Avenir's completeness guarantees are configuration-dependent, meaning that it may be the case that for a given pair of pipelines, Avenir successfuly maps one configuration σ, but fails to map another, σ'.

Relational Hoare Lenses Finally, in Chapter 7, we presented Relational Hoare Lenses, a general purpose framework for reasoning about synchronizing pairs of configurable programs. The Relational Hoare Lenses framework combines insights from Relational program logic and symmetric lenses, to derive a set of simple combinators that permit elegant and ergonomic synchronization programs that *carry* along their equivalence proofs. We replicate an experiment from Chapter 6 to prove that lenses synchronize pairs of pipelines while preserving equivalence.

9.1 Future Work

Cost-Aware Network Models The synthesis and semantic approaches to pipeline synchronization do not capture any information about the cost of various resources. At a simple level, hardware resources that implement match-action tables are often of a fixed-size. From the control-plane's perspective that means that they can only support a finite number of rules. However, both Avenir and our pipeline models in the work on Relational Hoare lenses ignore this detail, assuming that the hardware can support as many rules as is required to implement the abstract functionality. Further, different kinds of hardware, such as content-addressable memory (CAM) and ternary CAM (TCAM) have different cost requirements, e.g. in terms of time spent to insert rules and power required to use them. Building synthesis and verification tools that preserve or minimize cost models here is important for providing robust abstract interfaces for network devices.

Relational Synthesis Further, for certain networking tables that require rapid and predictable re-configuration, the overhead of invoking a dynamic synthesis loop like Avenir will greatly limit the ability of the network to react to changes. Instead, we would like to synthesize static code that can be deployed in the control that realizes these control plane mappings. One potentiall way to do this would be to synthesize relational hoare lenses, which would synthesize altogether a control plane mapping lens, a proof of equialence, and the ci-specs required to make the proof of equivalence (and the lens laws) hold. We believe that the antiunification observations we made in Avenir's template and theorem caches could provide the foundation for such an algorithm.

New Applications for Specificaiton Synthesis Finally, the techniques that we've applied here are relatively general, and can be applied across domains. Capisce's algorithms for ci-spec inference should be portable to other domains that admit quantifier elimination, for instance computing data integrity constraints for a database management system (DBMS). The ability to efficiently monitor the computed specifications could be useful in runtime-monitoring systems that are relatively un-verifiable, such as artificial intelligence systems.

New Applications for Relational Hoare Lenses The work on RHLenses has a very broad set of potential applications. For instance, it has potential applications in compiling between interpreted langauges. The lens would represent compilers between the two languages, and the relational verification would prove equivalence between the interpreters or runtime systems. Moving in a different direction, RHLenses could be used as a framework for verifying database migrations—the databases instances represent the schema and the programs are the database applications themselves. In a general sense, the RHLenses provide a robust and general framework for building verified software across many domains.

BIBLIOGRAPHY

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. Switchv: Automated sdn switch validation with p4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 365–379, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 789–801, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. *ACM SIGPLAN Notices*, 51(1):789–801, 2016.

[5] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, January 2005.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 113–126, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. Bit-vector interpolation and quantifier elimination by lazy reduction. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

[8] Jiasong Bai, Jun Bi, Menghao Zhang, and Guanyu Li. Filtering spoofed IP traffic using switching ASICs. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 51–53. ACM, 2018.

[9] Gilles Barthe. An introduction to relational program verification, 2020.

[10] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM*

2011: Formal Methods, pages 200–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 29–43, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):847–859, 2016. Articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday.

[13] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. An assertion-based program logic for probabilistic programs. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, pages 117–144, 2018.

[14] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 437–451, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 14–25, New York, NY, USA, 2004. Association for Computing Machinery.

[16] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004.

[17] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 969–984, 2020.

[19] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, page 338–347, New York, NY, USA, 2006. Association for Computing Machinery.

[20] 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 Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[21] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. pages 99–110, August 2013.

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

[23] Eric Hayden Campbell, Hossein Hojjat, and Nate Foster. Computing precise control interface specifications. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):905–934, 2024.

[24] John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proceedings of the ACM on Programming Languages*, 2(POPL):15, 2017.

[25] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[26] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 342–356, August 2018.

[27] P4 Language Consortium. P4Runtime. `https://p4.org/p4-runtime/`, October 2017. Accessed March, 2021.

[28] P4 Language Consortium. P416 Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf`, 2018. Accessed March, 2021.

[29] P4 Language Consortium. P4 16 language specification v.1.2.2. `https://p4.org/p4-spec/docs/P4-16-v1.2.2.html`, May 2021.

[30] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[32] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM (CACM)*, 18(8):453–457, August 1975.

[33] Isil Dillig and Thomas Dillig. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*, pages 684–689. Springer, 2013.

[34] Isil Dillig, Thomas Dillig, and Alex Aiken. Reasoning about the unknown in static analysis. *Communications of the ACM*, 53(8):115–123, 2010.

[35] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *SIGPLAN Not.*, 48(10):443–456, oct 2013.

[36] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. Bf4: Towards bug-free p4 programs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 571–585, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Matthias Eichholz, Eric Hayden Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. How to Avoid Making a Billion-Dollar Mistake: Type-Safe

Data Plane Programming with SafeP4. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[38] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. Dependently-typed data plane programming. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[39] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[40] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-Wide Configuration Synthesis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, Heidelberg, Germany (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 261–281, 2017.

[41] fabric.p4 source code. `https://github.com/opennetworkinglab/onos/blob/2.2.2/pipelines/fabric/impl/src/main/resources/fabric.p4`, 2022. Accessed 2022.

[42] fabric.p4 source code from ONOS v2.2.2. `https://github.com/opennetworkinglab/onos/blob/2.2.2/pipelines/fabric/impl/src/main/resources/fabric.p4`, 2020. Accessed March, 2021.

[43] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[44] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[45] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[46] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: gen-

erating compact verification conditions. volume 36, page 193–205, New York, NY, USA, jan 2001. Association for Computing Machinery.

[47] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK (POPL)*, pages 193–205, 2001.

[48] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17–es, May 2007.

[49] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 383–396, New York, NY, USA, 2008. Association for Computing Machinery.

[50] John Nathan Foster. *Bidirectional programming languages.* PhD thesis, University of Pennsylvania, 2009.

[51] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN International Conference on Functional Programming, Tokyo, Japan (ICFP)*, pages 279–291, September 2011.

[52] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. Using deep programmability to put network owners in control. *SIGCOMM Comput. Commun. Rev.*, 50(4):82–88, oct 2020.

[53] Xiangyu Gao, Taegyun Kim, Michael Dean Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation using Program Synthesis. In *ACM Special Interest Group on Data Communication, Virtual Event, USA (SIGCOMM)*, pages 44–61, August 2020.

[54] P4.org Architecture Working Group. P4 16 portable switch architecture (psa). `https://p4.org/p4-spec/docs/PSA.html`, Apr 2021.

[55] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

[56] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[57] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. *SIGPLAN Not.*, 46(1):371–384, jan 2011.

[58] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[59] Intel. Intel tofino series programmable ethernet switch asic.

[60] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA (NSDI)*, pages 87–101, May 2015.

[62] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[63] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[64] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, October 1983.

[65] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 637–650, 2015.

[66] Ali Kheradmand and Grigore Rosu. P4K: A Formal Semantics of P4 and Applications. *Computing Research Repository (CoRR)*, abs/1804.01468, 2018.

[67] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[68] Chaitanya Kodeboyina. An open-source P4 switch with SAI support, Jun 2015.

[69] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 351–364, October 2010.

[70] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theor. Comp. Sys.*, 59(2):323–376, aug 2016.

[71] Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, and Alceste Scalas. P4r-type: A verified api for p4 control plane programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[72] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[73] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A p4-based 5g user plane function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 162–168, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. Event-driven network programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA (PLDI)*, pages 369–385, 2016.

[75] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient

synthesis of network updates. volume 50, page 196–207, New York, NY, USA, jun 2015. Association for Computing Machinery.

[76] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

[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] Lambert Meertens. Designing constraint maintainers for user interaction. 07 1998.

[79] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Ling-Ling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. *SIGMOD Rec.*, 30(1):78–83, 2001.

[80] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL (NSDI)*, pages 1–14, April 2013.

[81] David A. Naumann. Thirty-seven years of relational hoare logic: Remarks on its principles and history. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, pages 93–116, 2020.

[82] Barefoot Networks. Behavioral model, Dec 2018.

[83] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SIGCOMM Symposium on SDN Research, Los Angeles, CA (SOSR)*, pages 5:1–5:7, March 2018.

[84] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, et al. Using P4 on Fixed-Pipeline and Programmable Stratum Switches. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Los Angeles, CA (ANCS)*, pages 1–2, October 2019.

[85] Peter O'Hearn. Separation logic. *Communications of the ACM (CACM)*, 62(2):86–95, January 2019.

[86] ONOS : Open Network Operating System. `https://github.com/opennetworkinglab/onos/commit/b7b79af9702f03c1286b8f2f9d98e6b87b29c467`. Accessed March, 2021.

[87] Open Compute Project. Switch abstraction interface, 2015.

[88] OpenConfig. `https://www.openconfig.net`. Accessed March, 2021.

[89] OpenDaylight. `https://www.opendaylight.org`. Accessed March, 2021.

[90] OpenFlow-Data Plane Abstraction Networking Software. `https://www.broadcom.com/products/ethernet-connectivity/software/of-dpa`. Accessed March, 2021.

[91] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, December 1976.

[92] Kanghee Park, Loris D'Antoni, and Thomas Reps. Synthesizing specifications. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[93] The POX OpenFlow Controller. `https://github.com/noxrepo/pox/`. Accessed March, 2021.

[94] QMX switches require the unicast flow being installed before multicast flow in TMAC table. `https://github.com/opennetworkinglab/onos/commit/45b69ab951915a4211a`. Accessed March, 2021.

[95] Lyle Harold Ramshaw. *Formalizing the analysis of algorithms.* PhD thesis, Stanford, CA, USA, 1979.

[96] John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, and Deian Stefan. Scooter & sidecar: a domain-specific approach to writing secure database migrations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 710–724, New York, NY, USA, 2021. Association for Computing Machinery.

[97] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh

Sivaraman, and Nate Foster. P4testgen: An extensible test oracle for p4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 136–151, New York, NY, USA, 2023. Association for Computing Machinery.

[98] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. NetGen: synthesizing data-plane configurations for network policies. In Jennifer Rexford and Amin Vahdat, editors, *ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, CA (SOSR)*, pages 17:1–17:6, June 2015.

[99] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 365–378, October 2010.

[100] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[101] SLOCCount. `https://dwheeler.com/sloccount/`. Accessed March, 2021.

[102] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A Fast Compiler for NetKAT. In *ACM SIGPLAN International Conference on Functional Programming, Vancouver, BV (2015)*, pages 328–341, August 2015.

[103] Armando Solar-Lezama. *Program Synthesis by Sketching.* PhD thesis, 2008.

[104] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *ACM SIGPLAN Notices*, pages 136–148, June 2008.

[105] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (ASPLOS)*, pages 404–415, 2006.

[106] SONiC. `https://azure.github.io/SONiC/`. Accessed March, 2021.

[107] Smolka Steffen, Ali Kheradmand, and Antonin Bas. p4lang/p4-constraints: Constraints on p4 objects enforced at runtime.

[108] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 518–532, New York, NY, USA, 2018. Association for Computing Machinery.

[109] Stratum: enabling the era of next generation SDN. `https://www.opennetworking.org/stratum/`. Accessed March, 2021.

[110] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 572–585, New York, NY, USA, 2017. Association for Computing Machinery.

[111] switch.p4 source code. `https://github.com/p4lang/switch`, 2020. Accessed Feb, 2022.

[112] 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 *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 17–32, New York, NY, USA, 2021. Association for Computing Machinery.

[113] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

[114] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in Software-Defined networks. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA, April 2012. USENIX Association.

[115] Trellis platform brief. `https://www.opennetworking.org/wp-content/uploads/2019/09/TrellisPlatformBrief.pdf`. Accessed March, 2021.

[116] v1model.p4 source code. 2021. Accessed Feb, 2022.

[117] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[118] Konstantin Weitz, Stefan Heule, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. Leveraging P4 for Fixed-Function Switches. In *P4 Workshop 2019*, 2019.

[119] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16–es, May 2007.

[120] Menghao Zhang. Anti-spoof, Nov 2018.