Lucid: A High-Level, Easy-To-Use Dataplane Programming Language

Devon Kennedy Loehr

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David Walker

January 2024

Abstract

The introduction of programmable switches and the P4 language for programming them has made it possible for network operators to write ever-more-sophisticated network applications, and execute them at high speed. However, *possible* is not the same as *easy*. The *de facto* standard dataplane programming language, P4, is akin to an "assembly language" for programmable switches: it provides a powerful but low-level interface to switch hardware. While this gives the programmer fine-grained control, it makes it difficult to write and reason about programs with complicated high-level behavior. Furthermore, modern hardware is heavily specialized for packet processing. While this specialization allows blazing fast speeds, it also heavily restricts the sorts of programs a programmable switch can run. Unfortunately, such restrictions are poorly represented in modern dataplane languages. Should a program violate one or more of these restrictions, it will fail to compile, often with an arcane or unhelpful error message.

This dissertation presents Lucid, a high-level, event-based language for programmable switches. Lucid raises the level of abstraction for data-plane programming in several ways. Control flow is represented by *events*, each of which has an associated *handler* that is executed when the event is generated. Different threads of control may be interleaved easily by writing them as handlers for separate events. Lucid provides high-level representations of switch hardware restrictions, which are enforced by syntactic checks, including a novel type system for detecting ordering violations.

Lucid is compiled to P4 for the Intel Tofino, and we find that Lucid programs typically contain around 10 times fewer lines of code than equivalent P4 programs. Furthermore, the Lucid interpreter may be used to quickly evaluate different configurations of a Lucid program, in order to automatically optimize a single program for different networking environments.

Acknowledgements

I would like to thank my advisor, David Walker, for taking me under his wing and supporting me through my graduate career. He has been an invaluable source of inspiration and advice, and I am extremely fortunate to have worked with him.

Next I would like to thank my collaborator, John Sonchack, without whom Lucid would not exist. Our numerous hour-long discussions are always fruitful and led to the language we have today. I can only hope to work with similar collaborators in the future.

I would like to thank as well my undergraduate adviser, Zachary Palmer, for introducing me to the realm of programming languages, providing me with early research experience, and supporting me in my graduate school applications.

Finally, I would like to thank my parents, without whom *I* would not exist. Their support, love and trust, have enabled me to pursue my education through graduate school, and for that I am endlessly grateful.

Contents

Chapter 1

Introduction

Every day, billions of people wake up and connect to the internet, be it to check their mail, communicate with friends, work remotely, or just look something up. We live in an increasingly interconnected and globalized world, in which time zones are a greater barrier to communication than planetary-scale distances. The very fabric of modern society is built upon computer networks, from those in individual homes and offices, to giant ISP backbones, to the myriad datacenters of cloud service providers.

As networks grow, the demands for speed and functionality grow with them. Early networks merely had to forward messages; modern networks contain a bevy of additional functionality, often including traffic management (e.g. load balancing [5, 45, 38]), security (firewalls, DDoS protection [19, 69, 50, 47]) and telemetry (monitoring, measurement [24, 89, 23, 10, 74]) With all this complexity, configuring and running a large-scale network is a herculean task.

Historically, much of this functionality was implemented via dedicated hardware, called **middleboxes**, which were inserted into the network between switches. However, in recent years there has been a shift from static, fixed-function devices to programmable hardware whose behavior can be customized, enabling not just configuration but the ability to run entirely new types of networking algorithms with

the same hardware. This approach is referred to as **Software-Defined Networking (SDN)**, and the art of programming network hardware to directly run new algorithms (as opposed to offloading that computation to a controller) is called **dataplane programming**.

The *de facto* standard language for dataplane programming is called P4 [14]. Since its introduction in 2015 [58], P4 has been the basis for a wide variety of dataplane programming research [2], covering all the previously-mentioned functionality and more. Modern programmable networking hardware increasingly supports P4, from switches like the Intel Tofino [1] to smartNICs like the Pensando DSC [8].

Despite P4's success, however, dataplane programming remains a challenging task. Programmable networking hardware is heavily optimized for its primary purpose: processing *many* packets, *fast*. As a result, the hardware places many restrictions, both implicit and explicit, on the sorts of programs it can run. What's more, P4 operates at a low level of abstraction, forcing dataplane programmers to reason about hardware primitives rather than the high-level behavior of their program. This makes it easy to run afoul of the hardware restrictions, and makes writing sophisticated programs prohibitively difficult.

For example, consider the simple application of a MAC learner, which automatically learns how to forward packets to different MAC addresses. Its high-level logic is easy to describe: it should maintain a mapping of MAC addresses to ports on the switch. When a packet arrives, it should add that packet's source MAC address to its map (if necessary), then look up the destination MAC address and forward the packet out of the appropriate port. If it does not have an entry for the destination, it should broadcast the packet on all ports.

However, writing even a simple MAC learner in P4 is challenging. To do so, one must:

1. Write a packet parser (as a state machine) to extract the relevant MAC addresses from the packet's headers.

2. Allocate memory to store the address map (using hashed MAC addresses as keys)[1].

3. Explicitly hash both the source and destination MAC address for each packet, and create specialized memory access actions to look them up.

4. If the source MAC is not in the address map, generate a new packet at the beginning of the switch (**recirculation**) to add it to the map[2].

5. Manually ensure that all accesses to the address map occur in a consistent order[3].

6. Send the packet out of the correct port(s).

Few of the steps above are simple! Indeed, they serve to illustrate two of the most pernicious hardware constraints. Steps 2-5 all deal with the complexities of **stateful memory management** on programmable hardware, which have significant restrictions on how often they can be accessed per packet, and in which order. As a result, step 4 requires **recirculating** the packet, sending it back to the beginning of the switch for additional processing (and additional memory accesses). Manually managing recirculation is a difficult process that involves configuring the switch ahead of time, as well as modifying every part of the program to account for the fact that recirculated packets should be handled differently.

[1] In practice, one must allocate two copies of the map, since we need to access it twice per packet.

[2] Note that this increases the number of different kinds of packets this program has to handle, which requires additional configuration.

[3] This ensures that the program can be laid out in the hardware's linear packet processing pipeline.

1.0.1 Existing Work

In the past decades, researchers have attempted to address many of the shortcomings of P4. Lyra [31] is a high-level dataplane programming language that allows users to program multiple switches at once. Domino [70] and Chipmunk [32] introduce abstractions for the computation capabilities of different pieces of hardware. P4All [36] introduces the ability to create flexibly-sized data structures which are automatically allocated memory by the compiler, rather than the user.

Each of these works provides useful insights, but each has drawbacks as well. Lyra provides a useful high-level programming interface, but relies entirely on synthesis for its compilation, meaning that users get no useful feedback should their program fail to compile. Domino and Chipmunk are able to abstract different types of hardware, but share the issue of synthesis-based compilation, and provide little ability to automatically optimize the programs they compile. P4All provides this capability, but requires significant user input to its optimizer, and operates at the same low level of abstraction as P4.

1.1 Lucid: A Language for Control in the Data Plane

The goal of this thesis is to improve the existing dataplane programming infrastructure by introducing **Lucid**, a high-level dataplane programming language emphasizing ease-of-use. Lucid is built on top of P4[4] but provides a different set of primitives and language features designed specifically for dataplane programming. To help ensure programs compile, Lucid embraces a *correct-by-construction* programming model, in which hardware restrictions are explicitly represented and enforced by the language.

[4]That is, a Lucid program compiles down to a P4 program.

Event-based programming Lucid programs are based around the idea of responding to **events** that occur in the network, such as the arrival of a packet or the failure of a link. Each event is associated with a **handler**, a block of imperative C-like code that is executed when the event occurs. Handlers may generate more events, potentially at different locations in the network, allowing communication between switches and coordination of control tasks. The Lucid-to-P4 compiler automatically interleaves the processing of events, meaning that Lucid programmers can avoid the complexities of recirculation and write different "threads" of control separately.

Modular programming The design of Lucid encourages modular programming techniques, allowing users to reason about different parts of their programs independently. In addition to the way that events allow users to write their threads of control separately, Lucid also provides general-purpose (but non-recursive) functions, as well as a module system allowing users to define reusable libraries encapsulated behind user-written interfaces. Indeed, the Lucid authors have created a small library of commonly used dataplane structures, such as Bloom filters, count-min sketches, and arrays with bounds-checking.

Correct-by-construction There are many ways a dataplane program might fail to compile, not all of which are directly visible in the program. Many of these *implicit restrictions* center around the way that switches treat stateful memory. For example, such memory must be accessed in a consistent order across all parts of the program. In addition, the amount of computation allowed during state updates is limited in various ways. Lucid uses language design to guide users into satisfying these constraints, by (1) providing a simple, high-level guideline on how to access stateful memory, and (2) enforcing that guideline through the use of a type system and a series of syntactic checks that provide useful, actionable feedback when violations are detected.

1.2 A MAC learner in Lucid

Returning to our MAC learner example from above, the steps to implement the same program in Lucid are as follows:

1. Define several arrays (stateful memory constructs), which store the learned MAC addresses.

2. Create an event representing incoming packets, whose handler hashes the packet's MAC addresses and looks them up[5].

3. Create an event whose handler updates the learned addresses when necessary.

   (a) The implicit recirculation and interleaving of these events' handlers is handled automatically by the Lucid compiler.

   (b) The Lucid type system will automatically alert the user if the address map is accessed in an inconsistent order.

4. Send the packet out of the correct port(s).

As you can see, the process for writing a simple MAC learner in Lucid is not only shorter, but substantially simpler: the complicated reasoning about recirculation and stateful memory access have been offloaded to the Lucid compiler. Indeed, the entire program takes only 34 lines of Lucid code, shown in Figure 1.1. In comparison, the P4 code output by the Lucid compiler is *874* lines, nearly a hundred of which[6] are dedicated to simply defining the headers and metadata to be used in the rest of the program[7].

[5]Defining a parser is not necessary for this simple program; we are able to generate one automatically because the event's arguments precisely describe the data to read from the packet.

[6]95, to be precise.

[7]One might point out that the output of the Lucid compiler is perhaps more complicated than P4 code a human would write, which is true. However, we have found that in general Lucid code is 10x shorter than even hand-written P4 code, so the discrepancy here is not exceptional.

```
// Type of an ethernet header
type eth_t = { int<48> src_mac; int<48> dst_mac; int<16> ethertype; }
const int HASH_SEED = 7;
const int ARRAY_LEN = 512;

// True if we've learned this mac address
global Array.t<1> learned_macs_1 = Array.create(ARRAY_LEN);
// Second copy so we can access it twice.
// We need this because we need to look up two indices,
// but each array can only be accessed once per handler
global Array.t<1> learned_macs_2 = Array.create(ARRAY_LEN);
// 8-bit ports, 1024 entries in the table
global Array.t<8> mac_table = Array.create(1024);

event learn_mac(int<48> mac, int<8> out_port) {
  int<9> hashed_mac = hash<9>(HASH_SEED, mac);
  Array.set(learned_macs_1, hashed_mac, 1);
  Array.set(learned_macs_2, hashed_mac, 1);
  Array.set(mac_table, hashed_mac, out_port);
}

event ethernet_packet(eth_t eth) {
  int<9> hashed_src = hash<9>(HASH_SEED, eth#src_mac);
  int<9> hashed_dst = hash<9>(HASH_SEED, eth#dst_mac);
  // Lookup src, and learn its mac address if necessary
  if (Array.get(learned_macs_1, hashed_src) == 0) {
    generate learn_mac(eth#src_mac, (int<8>)ingress_port);
  }

  // Lookup dst, and forward/flood the packet as appropriate
  if (Array.get(learned_macs_2, hashed_dst) == 1) {
    int<8> port = Array.get(mac_table, hashed_dst);
    generate_port (port, this);
  } else { // Flood the packet
    generate_ports (flood ingress_port, this);
  }
}
```

Figure 1.1: A simple MAC learner implemented in Lucid. To keep things simple, it does not handle hash collisions.

1.3 Contributions

This thesis makes the following contributions:

1. **Lucid** (Chapter 3), a new, high-level, event-driven dataplane programming language with an emphasis on *ease-of-use* and *correct-by-construction* code. Lucid programs are typically 10x shorter than corresponding P4 code, and are both modular and reusable.

2. **Pipeline Types** (Chapter 4), a novel type-and-effect system for tracking accesses to data laid out along a pipeline. We formalize the type system and prove its soundness, as well as describe its implementation for Lucid.

3. The **Lucid compiler** (Chapter 5), which translates a Lucid program into an equivalent P4 program that is specialized for the Intel Tofino. We implement a scheduling algorithm for statements in the Lucid program, and show that by using it we can compile much more sophisticated programs than with the Tofino compiler alone.

4. **Parasol** (Chapter 6), a framework for writing parameterized dataplane programs and automatically optimizing their high-level behavior. Parasol leverages Lucid's expressivity and built-in interpreter to optimize a wider range of parameters and objective functions than existing optimization frameworks.

1.3.1 Attribution

This work was supported in part by grants from the Network Programming Initiative and the National Science Foundation (1837030, 2223515, 2312539, and 1703493). The Lucid language as a whole was developed by the author and John Sonchack, with frequent advice and support from David Walker and Jennifer Rexford. In addition,

the work on Parasol was led by Mary Hogan, with contributions from Shir Landau-Feibish. A paper describing the language and some of its compilation process was published in SIGCOMM'21 [73]. A paper describing the type system was published in POPL'22 [51]. A paper describing Parasol is currently in submission [37]. More specific attributions are included at the beginning of each chapter.

Chapter 2

Background

This chapter is dedicated to explaining some basic principles of networks, dataplane programming and programmable switches. The concepts here will be referred to freely throughout the rest of the thesis. The structure of the chapter is as follows: first, we introduce the very basic concepts of modeling computer networks (§2.1), then describe the programmable switch architecture that Lucid targets (§2.2). Finally, we discuss how these switches are programmed today, and what makes modern dataplane programming difficult (§2.3).

2.1 Modeling Computer Networks

A network is fundamentally a *communication system* – its primary goal is to transfer messages from a **source** to a **destination**. The fundamental unit of a network is the **switch**: a device that forwards messages between two or more directly-connected **neighbors**. Each switch has many **ports** which are used to identify connections to neighbors. Switches can be "chained" to send messages across further and further distances.

This view of networks is naturally modeled as a graph, where nodes represent switches, and edges represent **links** (direct connections between switches). A par-

ticular model may be more or less complex depending on purpose; for example, a routing algorithm may use edge weights to represent bandwidth, transmission delay, price, or other properties of individual links. For the purposes of this thesis, we model networks as simple, unweighted, undirected graphs.

2.1.1 Packets and Headers

Since messages may be arbitrarily long, they are not transmitted through the network wholesale. Instead, individual messages are broken up by the source into smaller pieces called **packets**, which are transmitted individually. The destination is then responsible for reacting to these packets; depending on the communication protocol, this might involve reassembling them into the full message, responding to them individually, or something else.

In addition to the actual message (the **payload**), packets carry metadata in the form of **headers**. A header is a structured collection of data that is prepended to the payload. The first headers are added by the source, and contain important information such as the destination (so switches know where to send the packet). As the packet travels through the network, switches may add additional, temporary headers by prepending them to the existing headers. The collection of headers on a packet is sometimes called a **header stack**.

Common Protocols The format of a header is dictated by the packet's **transmission protocol**. In this thesis, we will refer to the following standard protocols:

- The **Ethernet** protocol is used for transmitting packets from switch to switch. Ethernet headers contain source and destination fields (both of which are hardware MAC addresses), as well as an **ethertype** field which indicates what type of header is next in the stack.

- The **Internet Protocol** (IP) is used for transmitting packets across networks

such as the internet. Like ethernet headers, IP headers contain source and destination fields, both of which are IP addresses. IP headers also contain source and destination **port numbers**: these have no relation to ports on a switch, but rather function as secondary addresses, directing packets to specific processes on the source/destination machines.

- The **Transport Control Protocol** (TCP) is a way of providing reliable, in-order packet delivery across networks, despite the fact that packets might get lost or re-ordered during transmission. TCP headers contain sequence numbers and checksums to ensure packets arrive in-order and uncorrupted.

2.1.2 Forwarding and Routing

When a packet arrives at a switch, that switch must make an important decision: where (i.e. to which neighbor) should it send that packet? This process is known as **forwarding**, and, due to the enormous volume of packets in modern networks, it is imperative that each packet be forwarded as quickly as possible. Accordingly, switches typically precompute a **forwarding table** that dictates which neighbor leads to which destination. Forwarding can then be accomplished by simply inspecting the packet's headers for its destination, consulting the precomputed table, and sending the packet to the corresponding neighbor.

The process of computing a forwarding table is called **routing**. Unlike forwarding, routing requires a wider view of the network, in order to figure out which neighbors can reach which destination. Routing should also be reactive to changing network conditions (e.g. a new switch is added, or a link suddenly fails). However, routing is not as time-critical as forwarding.

2.1.3 Data Plane and Control Plane

The split between forwarding and routing illustrates a key conceptual distinction between a network's **data plane** and **control plane**. The term "data plane" (or "dataplane") refers to the parts of a network that run at high speed, directly on optimized switch hardware; forwarding is the quintessential dataplane task. In contrast, the "control plane" is the part of the network that configures the data plane, and takes care of more complex and less time-critical tasks such as routing.

Typically, control-plane tasks run on a general-purpose CPU, which may or may not be physically located on a switch. As a result, such tasks are orders of magnitude slower than dataplane tasks, which run at "line rate" directly on the switch's packet-processing hardware. The tradeoff is that CPUs are more flexible, and hence can perform operations that are infeasible or impossible on hardware that has been optimized for packet processing.

2.1.4 Common Applications

Recent years have seen several developments in **dataplane programming**; that is, writing programs which run in the data plane, as opposed to using fixed-function, preconfigured hardware. These may be used either to introduce new kinds of functionality, or to move algorithms out of the control plane into the data plane. In the latter case, those algorithms benefit from the much faster speeds and lower latency offered by the data plane, but only if they can be made to work within the more restricted packet processing hardware.

Routing algorithms Typically, the data plane handles forwarding while the control plane handles routing decisions. However, network conditions often change quickly – links might go up or down, and different routes may see higher or lower traffic. In comparison to the speed of traffic ("line rate"), the control plane is very slow

to respond to these sorts of changing conditions. A dataplane routing algorithm like Hula [45] or Contra [38] can allow switches to make intelligent forwarding decisions at line rate, reacting to changing conditions orders of magnitude faster.

Middleboxes Although networks provide the abstraction of simple packet delivery, they often contain additional functionality such as traffic filtering (firewalls), transformation (Network Address Translators (NATs)), traffic manipulation (load balancers), and security (Intrusion Detection Systems (IDSs) or DDos Protection). Typically, this functionality is provided by a specialized piece of hardware inserted into the network between switches (a **middlebox**), though some might also be implemented in the control plane. Both approaches have downsides: as before, the control plane is slow to react, while middleboxes must be bought and maintained separately, and perform a specific task with limited ability for configuration. Moving this functionality into a programmable dataplane allows network operators to solve both problems.

Telemetry A crucial aspect of operating a modern network is *measurement*. Network operators are always looking for a more granular or detailed view of their network so they can understand its behavior. This knowledge might be used either to debug problems or to optimize packet processing. Dataplane programs provide a natural avenue for taking these measurements: they inherently operate at line rate, have direct access to the fields of a packet, and can perform stateful operations like aggregation to provide meaningful statistics to network operators.

Novel applications The advent of programmable dataplanes has seen the introduction of entirely new kinds of functionality. For example, NetCache [41] allows a programmable switch to act as a cache for a database server, by storing popular queries and responding directly to clients when they send a query it has cached. By allowing the network itself to take on this functionality, the database operator can

reduce load on their servers while also potentially decreasing their average response time.

2.2 PISA Switches

Many modern programmable switches, including the Intel Tofino [1], have adopted the **Protocol-Independent Switch Architecture** (PISA), in which packets are processed by a pipeline of stages, each of which does a small piece of work before passing the packet to the next stage. The pipeline is configurable, allowing users to reprogram its behavior. Importantly, the pipeline processes packets at a fixed rate, regardless of the program it's running. As a result, optimizing PISA programs is not a matter of ensuring that they run *faster*, as it often is for regular programs; instead, the goal is to make sure the program fits inside the pipeline in the first place.

2.2.1 PISA Overview

At a high level, a PISA switch consists of two **pipelines** with a **traffic manager** (TM) between them. The **ingress pipeline** processes incoming packets, chooses where (if anywhere) they should be sent, and sends them to the traffic manager. The TM is responsible for queuing packets while they wait to be forwarded, and selecting which one to forward next. Finally, the **egress** pipeline processes packets on their way out of the switch, though the available operations are more restricted (e.g. it cannot choose which port to send the packet out of, since that was already determined in ingress).

Both pipelines share a common structure, depicted in Figure 2.1: they begin with a **parser**, which translates incoming packets into structured data. This data is stored in special header fields called **Packet Header Vectors**, which are discussed more in §2.2.4. The parsed data is passed to a series of **stages**, which perform the actual

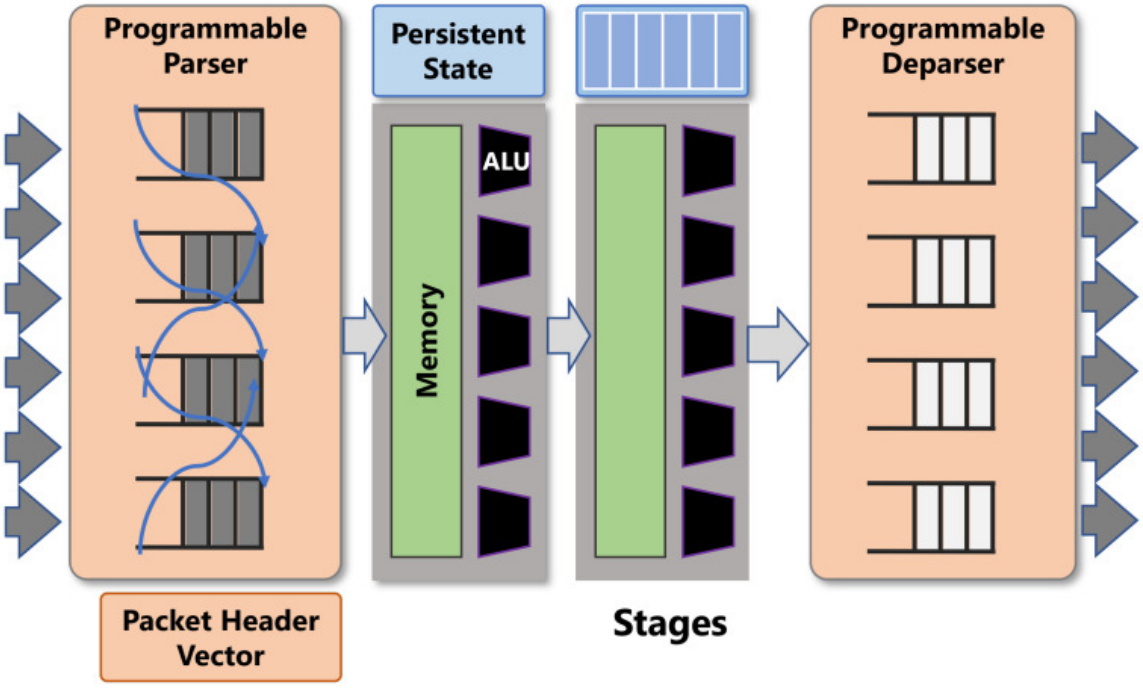

Figure 2.1: A high level overview of a PISA pipeline, taken from Butun et al. [17]. It demonstrates the most important components: the parser and deparser, and the processing stages in the middle, each of which has some persistent memory (at the top), and ALUs for performing computation at that stage.

packet processing. Finally, the pipeline ends by sending the packet to a **deparser**, which transforms the packet's structured data back into a stream of bytes.

2.2.2 Parsers and Deparsers

Like the rest of the pipeline, parsers and deparsers are programmable, and must be configured to parse the types of packets traveling through the network. This requires the user to define the set of possible headers in advance, along with rules for extracting those headers from the packet's raw bytes. Usually, not all of the packet will be parsed; the remaining bytes are called the **payload**[1]. Since the payload is unstructured, the pipeline does not operate on it, and it passes through unchanged. The parsed headers are passed to the first stage of the pipeline, along with metadata such as the time the packet came in and on what port.

The inverse operation is applied at the end of the pipeline. During processing, headers may be added, removed, or modified. When processing finishes, the deparser simply emits each header as a stream of bytes, followed by the payload. Note that the outgoing packet may have significantly different headers from the incoming packet.

2.2.3 Packet Processing Pipelines

Between the parser and deparser, PISA pipelines are comprised of a series of **stages**. Packets move forward through the pipeline, spending one clock cycle at each stage. At most one packet can be at each stage, and there is no direct communication between the stages apart from the packet itself. This means the pipeline is atomic: although multiple packets may be in the pipeline simultaneously (each at a different stage), the pipeline's observable behavior is always the same as if the packets were sent through one-at-a-time.

[1]Note that this is a slightly different definition of "payload" than before – in this case, the payload might include several layers of unparsed headers, as well as the packet's actual message.

Anatomy of a stage Each stage has several components, the most crucial of which are depicted in Figure 2.1. Each stage has many Arithmetic Logic Units (ALUs), which can be used to perform simple binary operations on integers. The stage also contains a certain amount of persistent memory called **registers**, as well as a small number of stateful ALUs (sALUs), which are capable of reading and writing that memory. Registers are crucial for dataplane programming because they are the only information that persists between packets; any stateful program must store its state in the registers.

Each register can only be accessed by one sALU per clock cycle. Since packets spend only one clock cycle at each stage, this means that a single packet cannot access the same register twice. Each sALU provides the ability to both read and write in a single clock cycle, and can perform some very restricted computation in between. However, several common design patterns are disallowed such as reading some data, performing a *series* of manipulations, and writing the result back to memory later. To implement these, one could send a packet back to the beginning of the pipeline to be processed again (**recirculation**), however, this is very expensive[2].

On the Tofino On the Intel Tofino, each stage also contains several **hash units** which can perform CRC hashes with a polynomial seed. Furthermore, each stage has a small number of **gateway tables** at the beginning of the stage, which are capable of performing very limited boolean computation (comparing a variable to another variable or a constant). Although all ALUs and sALUs operate simultaneously within a stage, hash units and gateway tables execute beforehand, meaning their results can be used by ALUs in the same stage.

For example, normally the snippet

[2]See §2.2.5 for more information.

```
if(x < y) {
  z = z + 1;
}
```

would require two stages to execute: one to compare `x` and `y`, and another to increment `z`. However, if we implement the comparison using a gateway table, both operations can be fit into a single pipeline stage.

Match-action tables The behavior of a stage is defined by the **match-action tables** it contains. Match-action tables are used to change the behavior of the switch for different packets. The `match` statement below demonstrates the semantics of a match-action table.

```
match x, y with
| 17, 54 -> printf "Branch 1"
| 17, _  -> printf "Branch 2"
| _, 42  -> printf "Branch 3"
| _, _   -> printf "Default branch"
```

This statement matches the variables `x` and `y` against several **patterns**, testing to see if they have the designated values. The first branch will be executed if `x` and `y` have values 17 and 54, respectively. The underscore character is a wildcard character, which matches everything. Hence the second branch will be executed if `x` has value 17, regardless of the value of `y`; similarly, the third branch will be executed if `y` has value 42.

Patterns are matched from top to bottom, with only the first matching branch being executed. Hence if `x` is 17 and `y` is 54, only branch 1 will execute. The last branch is always a "default" branch, with all-wildcard patterns.

The only difference between `match` statements and match-action tables is that instead of matching on variables, match-action tables match on designated bits of the packet header, and patterns are provided for each bit – either the constant patterns 1 and 0, or the wildcard pattern `*`, which matches either value. In a match-action

table, each branch is called a **rule**, and the body of each rule is called an **action**.

Types of match-action table Some match-action tables can be modified by the control plane while the switch is running, adding or removing rules. Such tables are called **dynamic** tables, and are exposed to the control plane during runtime. Tables that cannot be modified are called **static** tables, and are not visible outside the program. Static tables are advantageous when compiling a program because the compiler is free to modify them (for example, by merging two tables), so long as doing so does not change the semantics of the program. Since dynamic tables are visible to the outside world, the compiler is unable to change them without changing the interface to the control plane.

Separately, a table that contains no wildcard patterns in any of its rules is called an **exact** table; otherwise, it is a **ternary** table. Exact tables are significantly cheaper in terms of memory – they can be implemented in the switch's plentiful supply of SRAM[3], while ternary tables must fit into the much more limited TCAM[4]. Putting these together, we have four types of match-action table, based on two parameters: static vs. dynamic, and exact vs. ternary.

Behavior of a stage Each stage contains several match-action tables, which simultaneously match on their designated header fields and execute their chosen action. Each action operates using a subset of the ALUs and sALUs; there is no communication between these, so each action's operations are necessarily independent of each other. Once the ALUs are done executing, the packet is passed on to the next stage of the pipeline (or the deparser, at the end).

[3]Static Random-Access Memory.
[4]Ternary Content-Addressable Memory.

2.2.4 Packet Header Vectors

While a packet is being processed, temporary data (such as local variables and parsed header fields) is stored in the packet itself, in temporary headers called Packet Header Vectors (PHVs). Each PHV is a cluster of **slots** of the same width (typically 8, 16, or 32 bits). The number and size of PHVs is the same for every packet. Local variables and parsed headers are allocated to PHV slots at compile time. Variables can be partitioned across slots: a 16-bit variable might be stored in a 16-bit slot, two 8-bit slots, or half of a 32-bit slot (perhaps with the other 16 bits going to a different variable).

During execution, ALUs and hash units read and write from PHV slots. Hash units may read and write from any slot to any other slot without restriction. However, both kinds of ALU may only read and write to slots *in a single cluster*. This imposes constraints on where variables are stored: if one wants to first add together `x` and `y`, then add `z` to the result, all three variables must be located in the same cluster. Furthermore, each PHV slot may only be accessed by one ALU per stage. This means that it can be dangerous to store multiple variables in a single slot, since only one of those variables can be operated on at a time.

2.2.5 Recirculation

The linear and finite nature of pipelines means that sophisticated applications may not be able to express all their logic in a single pass through the pipeline. This might happen because the program simply takes too many stages, but it might also happen because the program needs to access some registers multiple times, which is impossible in a single pass since each packet only spends one clock cycle at each stage.

To implement such behavior, a programmer can send a packet back to the beginning of the pipeline to be processed again, with different header fields that cause different actions to be taken at each stage. This is called **recirculation**, and it is a

powerful technique for dataplane programming. Recirculation is done by sending the packet to a special **recirculation port** on the switch, rather than (or in a addition to) an exit port. After arrival, the recirculated packet is parsed and processed like any other incoming packet.

Recirculation's power comes with harsh downsides, however. Each recirculated packet consumes bandwidth that could have been used to process traffic. If every traffic packet generates one recirculated packet, half of the switch's processing power will be devoted to processing those recirculations, effectively halving its bandwidth! Furthermore, recirculation is difficult to express in a low-level dataplane programming language like P4; though it can be done, it involves significant work and switch configuration, and is both tedious and error-prone.

2.3 Programming PISA Switches

2.3.1 P4

The *de facto* standard language for programming PISA switches is P4 [14]. The original version [58] of P4 was designed specifically for PISA switches, and includes many switch-specific constructs, like match-action tables, as primitives. The most recent version[59] covers additional architectures beyond PISA, such as SmartNICs.

P4 can be thought of as an "assembly language" for switches: it is a low-level language that provides direct representations of the underlying hardware, with relatively few abstractions. This allows programmers to fully exploit the power of programmable pipelines, and modern industry work on dataplane programs is almost exclusively done using P4. Indeed, the introduction of P4 was critical to enabling most modern dataplane programming research.

Unfortunately, P4's power comes at a price: it makes many of the same tradeoffs as traditional assembly languages. Although it is powerful, and allows the user a high

```
apply {
  if (hdr.ip.dstip == 1){
    tmp = reg1_r.execute(0);
    hdr.ip.dstip = reg2_rw.
    execute(0);
  } else {
    tmp = reg2_r.execute(0);
    hdr.ip.dstip = reg1_rw.
    execute(0);
  }
}
```

(a)

```
apply {
  tmp = reg1_r.execute(0);
  hdr.ip.dstip = reg1_rw.execute(0);
}
```

(b)

```
error: Table placement cannot make any more progress. Though some tables have
not yet been placed, dependency analysis has found that no more tables are
placeable. This may be due to shared attachments on partly placed tables; may
be able to avoid the problem with @stage on those tables
```

(c)

Figure 2.2: The p4 snippets in (a) and (b) both result in the same error – an ordering error – but no indication of where it comes from or how to fix it (the `@stage` suggestion will not help in this case)

degree of control, the lack of abstractions leads to verbose and difficult-to-understand code. Worse, the low-level nature of the language means that error messages are often confusing, and provide little help in diagnosing the root cause. For example, the P4 program snippets in Figures 2.2a and 2.2b both result in the same error message (Figure 2.2c), but do not explain where in the program the error comes from or how to fix it (the suggested `@stage` annotation will not help in this case). As a result, there are many sophisticated applications that *could* be implemented on the hardware, but which have never been written because doing so in P4 is just too frustrating.

2.3.2 Why is P4 programming hard?

The restrictions of the PISA architecture and the low-level nature of P4 combine to make sophisticated applications frustrating, tedious and error-prone to write.

Hardware restrictions The most fundamental challenge of dataplane programming is ensuring the programs fit within the limited resources of the switch, while executing in the limited environment of a pipeline. Data structures must fit in mem-

ory; operations must be assigned to ALUs, and all of this must fit within the dozen-or-so stages of the pipeline. The problem gets more complex when data dependencies are considered, which spread operations across multiple stages, stretching out the pipeline.

A particularly notable class of error that can arise is called an **ordering error**. These errors are a by-product of the linear nature of a pipeline, and are caused when different parts of a program wish to access two pieces of memory in opposite orders. For example, one function accesses register A and later register B; another function accesses B first, then A. Since the registers are laid out along the pipeline, these functions impose contradictory constraints on the compiler: the first requires A to appear in an earlier stage than B, while the second requires the opposite. Fixing an ordering error requires the code to be reworked so that all parts access memory in a consistent order.

Low-Level Primitives The example of ordering errors highlights another difficulty of modern dataplane programming: P4 is a very low-level language, and using it involves directly configuring different parts of the hardware. In the pipeline, this involves creating match-action tables, defining the headers fields they match against (be sure not to miss any!), and writing the actual computation in the form of individual actions. It also involves configuring parsers and deparsers to read and write the appropriate header fields, including metadata fields controlling the fate of the packet (e.g. its destination port).

One significant challenge that arises due to the low-level nature of P4 is implementing recirculation. In order to recirculate a packet, a user must (1) set the destination port to the switch's designated recirculation port, (2) set metadata flags indicating that the packet has been recirculated, plus any additional relevant information, and (3) update every part of the pipeline to check for that flag and perform a different

```
error: Table placement cannot make any more progress. Though some tables have
not yet been placed, dependency analysis has found that no more tables are
placeable. This may be due to shared attachments on partly placed tables; may
be able to avoid the problem with @stage on those tables
```

Figure 2.3: Output of the p4 compiler when presented with an ordering error.

```
error.bfa:71: error: Can't reference phv in first operand of add instruction
failed command assembler
```

```
error.bfa:68: error: Can't have more than one constant operand to an SALU
compare
failed command assembler
```

Figure 2.4: Possible outputs of the p4 compiler when presented with an invalid RegisterAction. Note that the p4 program contains no reference to phvs, SALUs, or instructions and their operands.

action if it is present. If multiple types of packets need to be recirculated, this process must be repeated for each of them. Missing a single step can cause mysterious and hard-to-debug errors where a single part of the pipeline is wrong, causing a cascade of non-obvious effects.

A similar issue arises when trying to interleave processing of different packet types. Each table must ensure that it is applying an action appropriate for the current packet, and getting it wrong in a single stage could cause a chain of bizarre errors.

Incomplete Compilation Modern dataplane languages also lack abstractions for hardware *restrictions*. Certain operations are fundamentally impossible to execute on a PISA switch in a single pass through the pipeline; memory accesses that lead to ordering errors are a good example. Other operations are limited in unclear or target-dependent ways. For example, the Tofino's sALUs are capable of performing a limited amount of work at the same time as they read or write memory. However, no language represents exactly what these limitations are; furthermore, different models of switch will have different limitations.

Synthesis-Based Compilation There has been substantial work on compilers for P4, and dataplane languages in general, which use synthesis techniques to generate

layouts for their programs. Indeed, all the dataplane languages described in the next section use some variant of synthesis for their compiler. Synthesis-based techniques are powerful – they can provide a guarantee that if the program is compilable, the compiler will succeed. However, these techniques also have a downside: if the program fails to compile, they typically provide little or no useful feedback about *why*. This makes it very difficult to figure out what's wrong with a particular program, much less how to fix it.

Poor Portability Different hardware implementations of a PISA Pipeline don't necessarily have the same expressive power[5]. Different switches have different restrictions and idiosyncrasies that constrain the programs they can support. This means that although P4 programs are theoretically portable (able to run on multiple different targets), in practice a program developed for one device will be specialized to that device's capabilities. Attempting to compile that program to a different target may or may not succeed, depending on whether that program happens to satisfy the restrictions of the other target as well.

Optimization Even once a program is written, compiled, and deployed, it may still not perform well in practice. Dataplane programs typically involve several parameters that govern their performance, often involving apportionment of the switch's limited resources (e.g. how much memory is allocated to each data structure). Varying these parameters can have drastic effects on the program's performance, but the relationship of each parameter to the final outcome is complicated. It is often difficult to predict how tweaking a single parameter will affect performance; determining optimal values for each of them is even harder.

Furthermore, PISA programs are optimized on different criteria than standard, general-purpose programs. Typically, a CPU program will try to minimize execution

[5]Unlike, say, Turing Machines.

speed and/or memory usage. In contrast, dataplane programs always execute at the same speed (the speed of the pipeline), and have fixed memory footprints (allocated at compile time). As a result, there is no point trying to make programs "go faster". Instead, dataplane optimization targets high-level behavior (e.g. the hit rate of a cache), as well as trying to ensure that the program fits into the pipeline in the first place (e.g. it does not require more stages than are available).

The standard method for optimizing dataplane programs is to compile the program, and test its performance either on a test switch or in a simulator. Unfortunately, both steps are slow: P4 compilation can takes anywhere from minutes to hours to *days* in extreme cases. Similarly, the proprietary Tofino simulator processes only 1 packet per second, an untenable speed when tests involve processing tens or hundreds of thousands of packets.

2.3.3 Other dataplane languages

Since the introduction of P4, researchers have developed several higher-level dataplane programming languages, each of which addresses some of the above issues.

Lyra Lyra [31] is a high-level language providing a "one big switch" abstraction to its users. A network operator writes their code as if it were being processed by a single switch with generic operations. The Lyra compiler then uses optimization and synthesis techniques to compile this program to several different switches in the network, allocating different parts of the functionality to different switches.

Lyra does a good job of raising the level of abstraction over P4, by providing generic primitives instead of switch-specific ones, and allowing users to reason about a single switch instead of many. However, when its compiler fails to synthesize a working program, the user receives no useful feedback. If a program contains an ordering error, or attempts to do too much computation while accessing registers,

they will not be informed of this, and will be unable to easily figure out how they should adjust their program. Furthermore, the level of abstraction could be raised higher, as it is still centered around switch pipelines and explicitly manipulating header fields. Finally, while Lyra's compiler automatically tries to ensure programs fit into switch pipelines, it does not help with the issue of optimizing the behavior of programs.

Domino, Chipmunk, and CaT Domino [70] is another high-level language which provides an abstraction called **packet transactions** that capture the application logic for processing a single packet. Transactions are compiled into a series of **atoms**, which are individual pieces of computation that a switch can process in a single stage. A program can also contain **guards**, which are predicates describing which transaction(s) to run on a given input packet.

Domino provides a higher-level view of the switch hardware than Lyra; transactions are represented as sequential imperative code, and the compiler automatically breaks them down into atoms (which correspond to actions in the switch's pipeline). Notably, Domino provides a way of representing differences in switch hardware: different switches can define different atoms that they support. However, the Domino compiler targets an abstract "Banzai" virtual machine model, rather than any physical hardware, and does not contain representations for important hardware restrictions such as ordering errors. Furthermore, the compiler uses synthesis to map transactions into sequences of atoms; this process may fail unhelpfully or take a long time. Domino also provides no easy solution for optimizing programs without deploying them.

Chipmunk [32] is an extension of Domino's compiler that introduces a new compilation technique called **slicing**. Slicing reduces compile times significantly, and increases the chance that synthesis will succeed. The Chipmunk compiler also provides the ability to compile to the Intel Tofino, a more practical target than the Banzai

machine model. However, Chipmunk is still synthesis-based, which comes with the same downsides of unhelpful errors upon failing and potentially long compile times.

CaT [33] is a compiler for P4 programs which addresses several of the downsides of synthesis-based compilation, by decomposing the compilation process into three phases (Transformation, Synthesis, and Allocation) based on the Domino/Chipmunk transaction framework. Each phase presents a smaller, and thus hopefully more tractable, synthesis problem. Doing so allows them to compile more programs than existing compilers, and do so faster. However, the phases are ultimately still synthesis-based, and thus inherit the same downsides as Domino and Chipmunk.

P4All P4All [36] is an extension of P4 that introduces the ability to declare flexibly sized data structures by defining **symbolic variables** to represent their memory allocation. For example, a series of arrays might have two parameters: one for the length of the arrays (in bits), and another for the number of arrays in the series. The user must then write a formula relating the size of the data structure to the performance of the program (e.g. the relation between the sizes of a multi-stage cache and its overall hit rate). This formula is optimized using Integer Linear Programming (ILP) to produce a program that both fits within the switch's pipeline, and has optimized high-level behavior.

Unlike the other languages, P4All provides a way to optimize programs before they are deployed. However, the optimizer requires a closed-form formula that can be solved via ILP; deriving such a formula requires significant theoretical work that is infeasible for most users. Furthermore, P4All does not provide any additional abstractions on top of P4, meaning that it inherits most of the downsides of P4, including tedious programming and unhelpful errors.

Chapter 3

The Lucid Language

Lucid is a high-level, event-driven PISA programming language whose fundamental goal is to *make dataplane programming easier*. As dataplane programs, Lucid programs run inside a switch's packet processing pipeline, but users need not worry about manually configuring those pipelines. Instead, Lucid code takes an imperative, C-like form, and comes with several mechanisms, including a novel type system, for helping ensure Lucid programs will compile. Lucid's high level of abstraction makes it accessible even to novices without detailed knowledge of hardware or network programming, and helps experts to write sophisticated applications that would be infeasible to express using a lower-level language like P4.

3.0.1 Attribution

The Lucid language syntax and features were designed collaboratively by the author, John Sonchack, and David Walker. A description of the language appears in the original Lucid paper [73]. The first section of this chapter is modeled on the description given there.

3.1 Lucid by Example

As an introduction to Lucid's main features, we will look at several simple Lucid programs: first a basic traffic processing/monitoring application, and then several increasingly sophisticated implementations of a Bloom Filter, a common networking data structure. The code we display below is actual Lucid code.

In this section, we will illustrate the following features of Lucid:

1. How to use **events** and **handlers** to define the behavior of a Lucid program.
2. How to use **persistent memory** to retain information about packets, and how to use that information to affect the processing of future packets.
3. How to use Lucid's **module system** to create an abstract interface for data structures, allowing different implementations to be swapped out seamlessly.
4. How to use **vectors** and **loops** to support data structures with sizes that are determined only at compile time.

3.1.1 Event-based Dataplane Programming

Throughout this section, we will refer to Figure 3.1, which shows a simple Lucid program that forwards IPv4 traffic while periodically sending heartbeats to a controller, and alerting the controller about any unexpected traffic.

The core abstraction of Lucid is that of an **event**. Each event represents something happening in the network, such as a packet arriving at a switch, a link failing, or receiving a request to perform a control task like updating a firewall. Whenever these happen, the corresponding event is **generated** at one or more switches. *The fundamental behavior of a Lucid program is to react to incoming events.*

The set of possible events handled by a Lucid program is declared by the programmer; in Figure 3.1, those events are `eth, unexpected_etherty, heartbeat`,

```
// The eth event represents an incoming ethernet packet. Its arguments are
// the 48-bit MAC destination and source headers, the 16-bit ethertype
// header, and an abstract payload representing the rest of the packet
packet event eth(int<48> dst, int<48> src, int<16> etherty, Payload.t p);
// This alerts the controller if we get an event with a non-IPv4 ethertype
event unexpected_etherty(int<48> src, int<16> etherty);

// Send regular heartbeats to the controller so it knows we're still alive
event heartbeat(int count);
event send_heartbeat(int count);

// Handlers specify what to do when an event arrives at the switch
handle eth(int<48> dst, int<48> src, int<16> etherty, Payload.t p) {
    // Alert controller if we get a non-IPv4 packet
    if(etherty != 0x0800) {
        // Create a value representing the alert event
        event alert = unexpected_etherty(src, etherty);
        // Send the event out the hardcoded port connected to the controller
        generate_port (CONTROL_PORT, alert);
    } else {
        // If everything's fine, forward the packet unchanged
        generate_port (OUT_PORT, eth(dst, src, etherty, p))
    }
}

// Send a heartbeat to the controller at regular intervals
handle send_heartbeat(int seq) {
    generate_port (CONTROL_PORT, heartbeat(seq));
    event next = send_heatbeat(seq + 1);
    // Send next heartbeat in 1000 ns
    next = Event.delay(next, 1000);
    // generating without a port sends the event to ourself
    generate next;
}

// We should never receive these events, so just drop them
handle heartbeat(int count) { skip; }
handle unexpected_etherty(int<48> src, int<16> etherty) { skip; }
```

Figure 3.1: A basic Lucid program that performs three functions: forwarding IPv4 traffic, reporting non-IPv4 traffic to a controller, and sending regular heartbeats so the controller knows it's still running.

and `send_heartbeat`. Each event carries data deemed relevant by the programmer. For example, `eth`, which represents an ethernet packet, carries the first three ethernet header fields, followed by the contents of the packet (the "payload"). On the other hand, `heartbeat`, which represents a message to the controller, carries only the information it wants to communicate – in this case, a sequence number, so that missed heartbeats can be detected by the controller.

Lucid distinguishes two kinds of events: **packet** events and **background** events. Packet events are a direct representation of an incoming traffic packet, and their arguments represent that packet's headers (and optionally its payload). All other events are background events, and their meaning is entirely defined by the programmer. They are so named because they operate asynchronously "in the background" while the switch is processing regular traffic. Common uses for background events are updating or cleaning persistent memory, or communicating among switches running the same Lucid program.

Handlers Every event has a corresponding **handler** (lines 13 and 27 in Figure 3.1), which defines the actions to take when that event arrives. Handlers are atomic – although a switch can process multiple events at the same time, the results are always the same as processing them individually.

Compilation Note: Events and Handlers Under the hood, every event corresponds to a packet, and every handler is executed in the switch's packet processing pipeline. Packet events are literal traffic packets, while background events are ethernet packets with a special ethertype value. In both cases, the event's arguments are represented as special header fields. The Lucid compiler automatically generates header formats and parsers for background events. For simple programs, the parser for packet events may be automatically generated; otherwise, it must be written by the user, as described in §3.2.10.

Generating events In addition to receiving events, Lucid programs are capable of **generating** new events. In the case of packet events, this corresponds to sending out a packet whose headers are determined by the event arguments. For background events, it means that an event with the given arguments appears at the destination.

Events always appear at a particular switch in the network, specified when the event is generated. Sending events to other switches is done via the `generate_port` keyword, which lets the user specify which port the event should be emitted from. Switches can also send events to themselves using the `generate` keyword; this can be useful for triggering sub-tasks or, as in Figure 3.1 (Line 34), for repeatedly running the same handler in a loop.

In some cases, the programmer might not want an event to be executed immediately. This is common for background tasks that should be periodically executed, as with the `send_heartbeat` event in Figure 3.1. To facilitate this, events can be *delayed* using the `Event.delay` function. When a delayed event is generated, it is instead added to a buffer, and is emitted only after its delay has elapsed.

Interleaving control tasks One of the great strengths of Lucid is its ability to easily write programs that perform several unrelated tasks simultaneously. The program in Figure 3.1 is a good, if basic, example: it combines the tasks of traffic processing, traffic monitoring, and sending heartbeats. Writing an equivalent program in P4 would be much more complicated. To give a brief summary, the user must

1. Distinguish which type of packet is being processed at a given time,
2. Take different types of actions at each stage based on packet type,
3. Configure the hardware to enable recirculation
4. Manually copy and annotate packets for recirculation, and

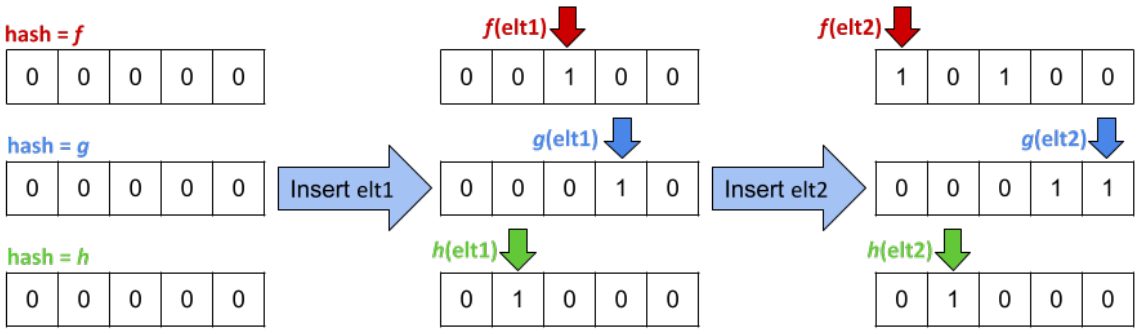

Figure 3.2: A Bloom filter with $k = 3$ and $m = 5$. Each array is associated with a hash function (f, g, or h), which produces an index; when an element is added, each array applies its hash function and sets the corresponding bit to 1.

5. Manually manage the hardware's packet queue to ensure delayed events are emitted at the right time.

In Lucid, all these tasks are handled automatically by the compiler. The programmer can add new threads of control to any program simply by declaring more events, without changing any existing code.

3.1.2 Bloom Filters

Bloom filters are a commonly used data structure in networking applications, forming the foundation of many dataplane algorithms. A Bloom filter is a probabilistic representation of a set, with two operations: adding an element, and checking membership. Membership queries may return false positives, but never false negatives.

On a switch, Bloom filters are typically implemented as a series of k bit arrays of length m, each of which is associated with a different hash function[1]. Each array element is initialized to 0. To add an element, that element is hashed with each function to produce an index into the associated array; that index is then set to 1. This process is illustrated in Figure 3.2.

[1]On a general-purpose computer, a Bloom filter may also be implemented as a single array of length m; however, doing so is infeasible on a switch due to the linear nature of the packet processing pipeline.

```
1  const int m = ...;
2  // Declare two 1-bit arrays with m entries each, initialized to 0
3  global Array.t<1> a0 = Array.create(m);
4  global Array.t<1> a1 = Array.create(m);
5  const int s0 = ...;  // Seed for first hash function
6  const int s1 = ...;  // Seed for second hash function
7
8  // Add item to filter
9  fun void add(int item) {
10     int idx0 = hash(s0, item);
11     int idx1 = hash(s1, item);
12     Array.set(a0, idx0, 1);
13     Array.set(a1, idx1, 1);
14 }
15
16 // Return true if item in filter
17 fun bool query(int item) {
18     int idx0 = hash(s0, item);
19     int idx1 = hash(s1, item);
20     int<1> b0 = Array.get(a0, idx0);
21     int<1> b1 = Array.get(a1, idx1);
22     return (b0 == 1 and b1 == 1);
23 }
```

Figure 3.3: A basic Bloom filter with k = 2. Functions `add` and `query` may be called from many different handlers.

Membership queries perform the same hashes, and return `true` if and only if each index was already set to 1. A false positive might therefore occur if all the indices happened to already be set to 1. The rate of false positives can be reduced by increasing k and/or m, at the cost of more memory usage.

3.1.3 Persistent Memory

Bloom filters are a persistent, stateful data structure; if one event adds an element to the filter, then future events should be able to observe that change. Lucid provides access to the switch's persistent memory in the form of **global values** (often referred to as simply "globals"). The most basic global value is an array of integers. Figure 3.3 shows a how arrays may be used to define a very basic Bloom filter program with $k = 2$.

Globals are declared using the `global` keyword, as shown in Figure 3.3 (lines

Figure 3.4: The output of the Lucid compiler when an ordering error is detected in a function. Note that the specific offending line is highlighted, allowing users to jump right to the appropriate part of their program for debugging.

3-4). Globals are always declared at top-level (never inside a handler), hence the name. The type of arrays in Lucid is `Array.t<sz>`, where `sz` is a **size** – an integer representing a bitwidth (in this case, the number of bits in each slot of the array). The program in Figure 3.3 creates two arrays with `m` 1-bit slots each, using the constructor `Array.create`.

Arrays can be accessed from handler bodies, using the `Array.set` and `Array.get` methods, as shown in Figure 3.3 (lines 12-13, 20-21). The `set` method takes the array, the index to modify, and the value to set it to; the `get` method takes only the array and the index to read from.

Ordering errors The nature of PISA pipelines means that there are several restrictions on the way arrays can be accessed in a single handler. Fortunately, Lucid's type system enables us to provide a simple, high-level description of the restrictions: specifically, *globals must be used in the order they are declared, and no more than once each.* As an example, it would be impermissible to swap lines 13 and 14 in in Figure 3.3, since that would result in using `a1` before `a0`, even though `a0` was declared first. Any operation that reads from or writes to an array counts as a "use" for this purpose.

We call this restriction the **ordering restriction**, and refer to the order of the global variables in a program as the **global order**. The ordering restriction must hold along every control path in the program separately, so it is permissible to, for

```
interface BloomFilterInterface {
    global type t;
    constructor t create(int m, int seed0, int seed1);

    fun void add  (t filter, int item);
    fun bool query(t filter, int item);
}
```

Figure 3.5: A simple interface for a Bloom Filter module

example, use the same global in both branches of an `if` statement[2]. Violating the global order results in an uncompilable program, and is referred to as an **ordering error**.

When an ordering error occurs in a Lucid program, our compiler generates an error message pointing the user to the appropriate part of their program, as shown in Figure 3.4. We describe ordering errors in more detail, as well as the way we detect them, in Chapter 4.

3.1.4 Modules and Records

The program in Figure 3.3 suffices to implement the basic behavior of a Bloom filter. However, the implementation is not at all abstract: all the components of the filter (the arrays and seeds) are declared globally at top level, and hence can be accessed by any part of the program. This makes it possible for some completely separate part of the code to modify the filter arbitrarily, potentially breaking its invariants (for example, zeroing out an index might lead to false negatives). Furthermore, the behavior of the filter is hard-coded; one would have to duplicate all the code if one wished to employ two filters simultaneously.

To provide abstraction and facilitate re-use, Lucid provides a simple module system patterned after OCaml's [57]. A Lucid module is a block of code defining types, functions and/or events. Every module has an **interface** that describes which parts

[2]Not all globals need to be used in a particular path; it is allowed to "skip over" globals, though doing so forfeits the opportunity to access the skipped globals.

of the module are visible to the outside world. Figure 3.5 shows a simple interface for a Bloom filter module; it declares an abstract global type `t` as well as the `add` and `query` functions.

The body of the module is shown in Figure 3.6. In order to present the abstraction of a single type representing a Bloom filter, the arrays and seeds are combined into a single compound record type, as shown on lines 3-8. Since this type is abstract, the module also provides a constructor, which can be called inside of global declarations to create values of that type. Finally, the `add` and `query` functions are largely the same as before; however, rather than always operating on the same two arrays and seeds, they take the filter as an argument, and use the record projection operator `#` to retrieve its component values.

This modularized version is much more convenient than the hardcoded one. Since the type is abstract, users can only access it via `add` and `query`, and hence cannot accidentally break any invariants. Furthermore, users can easily create multiple filters in a single program using the constructor, and access them with the same set of functions. Finally, if the user wishes to change the implementation of the filter, they can do so without worrying about breaking code that uses it. So long as the filter maintains the same interface and high-level behavior, different implementations can be swapped out seamlessly.

3.1.5 Vectors and Loops

There is still one problem with our treatment of Bloom filters. Although swapping out different implementations is easy due to the increased abstraction, actually writing those implementations is likely to involve a significant amount of code duplication. For example, increasing k to 3 by adding an additional array and hash seed requires the programmer to redefine the type t, update its constructor, and write new `add` and `query` functions that expect exactly 3 arrays. In general, each value of k requires a

```
module BloomFilter : BloomFilterInterface = {
    // An abstract record type, with definition hidden from module clients
    type t = {
        array<1> a0;
        array<1> a1;
        int s0;
        int s1;
    }

    // A compile-time function for creating global values.
    constructor createFilter(int m, int seed0, int seed1) = {
        a0 = Array.create(m);
        a1 = Array.create(m);
        s0 = seed1;
        s1 = seed2;
    }

    // Add item to filter
    fun void add(t filter, int item) {
        int idx0 = hash(filter#s0, item);
        int idx1 = hash(filter#s1, item);
        Array.set(filter#a0, idx0, 1);
        Array.set(filter#a1, idx1, 1);
    }

    // Return true if item in filter
    fun bool query(t filter, int item) {
        int idx0 = hash(filter#s0, item);
        int idx1 = hash(filter#s1, item);
        int<1> b0 = Array.get(filter#a0, idx0);
        int<1> b1 = Array.get(filter#a1, idx1);
        return (b0 == 1 and b1 == 1);
    }
}

// Using the constructor
global BloomFilter.t f1 = BloomFilter.createFilter(...);
global BloomFilter.t f2 = BloomFilter.createFilter(...);
```

Figure 3.6: An abstract, compound type for Bloom filters. The hash symbol # is Lucid's record projection operator.

completely new module whose code varies only in the number of arrays/seeds that are processed.

Lucid provides a solution through the use of **vectors**, which are fixed-length lists of values. The type of a vector contains its size – for example, the type `int[4]` is the type of length-4 vectors of integers. Figure 3.7 shows how the module and interface could be rewritten using vectors to allow for Bloom filters with arbitrary k.

In the rewritten version, the filter type `t` now takes a polymorphic size parameter `'k`, similarly to arrays (the apostrophe indicates polymorphism, à la OCaml). The definition of the type (lines 11-14), now contains exactly two fields, both length-`k` vectors that store the arrays and the seeds, respectively. Line 18 shows how polymorphic vectors (those whose size is a variable) can be initialized using vector comprehensions.

Operations on vectors are provided in the form of bounded for-loops, as shown in the `add` and `query` functions. Each loop contains an index variable and an upper bound. The size of each vector accessed with the index variable must exactly match that upper bound, in order to prevent out-of-bounds accesses. Unbounded for-loops are not feasible to implement, due to the linear nature of switch hardware.

3.1.6 Data structure libraries

We have now written a modular, flexible and re-usable implementation of a Bloom filter. As Bloom filters are an important building block for dataplane algorithms, this is quite useful! The module can easily be imported into a program using Lucid's C++-style `#include` directive, and used multiple times therein. Indeed, the Lucid repository on Github contains a small library of general, abstract and re-usable data structures that we have defined in this way, including a more sophisticated version of the Bloom filter code shown here. Over time, we hope this library will grow to be a useful resource for dataplane programmers, who currently find themselves frequently re-implementing the same behavior.

```
interface BloomFilterInterface {
    global type t<'k>;
    constructor t<'k> create(int m, int['k] seeds);

    fun void add  (t<'k> filter, int item);
    fun bool query(t<'k> filter, int item);
}

module BloomFilter : BloomFilterInterface = {
    // An abstract record type, with definition hidden from module clients
    type t<'k> = {
        array<1>['k] arrs;
        int['k] seeds;
    }

    // A compile-time function for creating global values.
    constructor createFilter(int m, int<'k> seeds) = {
        arrs = [Array.create(m) for m < k];
        seeds = seeds;
    }

    // Add item to filter
    fun void add(t<'k> filter, int item) {
        for (i < 'k) {
            int idx = hash(filter#seeds[i], item);
            Array.set(filter#arrs[i], idx, 1);
        }
    }

    // Return true if item in filter
    fun bool query(t<'k> filter, int item) {
        bool acc = true;
        for (i < 'k) {
            int idx = hash(filter#seeds[i], item);
            int<1> b = Array.get(filter#arrs[i], idx);
            acc = acc and (b == 1);
        }
        return acc;
    }
}

// Using the constructor
global BloomFilter.t<2> f1 = BloomFilter.createFilter(1024, [0; 1]);
global BloomFilter.t<3> f2 = BloomFilter.createFilter(1024, [2; 3; 4]);
```

Figure 3.7: A Bloom filter module using vectors and loops to allow instantiations with different values of k

```
types              τ ::= ...

identifiers        x ::= ...

declarations       d ::= event  x(τ0 x0, τ1 x1,...);
                       | handle x(τ0 x0, τ1 x1,...) { s }
                       | const τ x = e;
                       | global τ x = e;
                       | module x { ds }
                       | ...

declaration lists  ds ::= d | d ds

statements         s ::= s s
                       | τ x = e
                       | x = e
                       | if(e) { s } else { s }
                       | for(x < e) { s }
                       | ...

expressions        e ::= v
                       | x
                       | e <binop> e
                       | e(e, e, ...)
                       | ...

values             v ::= ...

programs           p ::= ds
```

Figure 3.8: A simplified grammar of the core syntax of Lucid.

3.2 Advanced Lucid

Not every feature of Lucid is needed to implement a Bloom Filter. Now that the reader is familiar with the basics of the language, we give a high-level survey of the other important Lucid features.

3.2.1 Grammar Overview

Figure 3.8 provides a high-level, simplified grammar for the Lucid language. A Lucid program is a series of **declarations**, which define events, handlers, modules, and globals. The body of each handler is a **statement**, defining that handler's control flow. Statements may be nested using the sequencing operator $s;\ s$. Each statement

contains one or more **expressions** representing computations, such as arithmetic and function calls.

The grammar in Figure 3.8 is meant to give an overview of the structure of a Lucid program. As a result, it includes only the most basic language features. The remaining features are described in the rest of this section.

3.2.2 Hashing

Lucid includes a hash operator, which is commonly used for computing array indices. The syntax is as follows:

```
hash<sz>(seed, arg_1, arg_2, ..., arg_n);
```

The above operation produces a `sz`-bit integer by hashing together `arg_1` through `arg_n` using a CRC hash with `seed` as the seed. The expression may take any number of arguments, which must be integers, booleans, or (possibly nested) compound types containing only integers and booleans. Providing the seed 1 performs the identity hash, which simply concatenates all the arguments together as strings of bits, and truncates the result to `sz` bits.

3.2.3 Printing

When using the interpreter, Lucid allows users to print strings to the terminal using the builtin `printf` statement, as shown below:

```
int count = ...;
printf("The current count is %d", count);
```

This function behaves much like `printf` in other languages, taking a string that contains zero or more **format specifiers** beginning with `%`, and one argument for each specifier. The arguments are substituted for the specifiers in order, and printed to the terminal. Lucid supports two specifiers: `%d` for integers and `%b` for booleans. Print

statements are only executed in the Lucid interpreter, and are ignored otherwise.

3.2.4 Event destinations

We have already seen how events can be sent out a specific port using `generate_port` or sent to oneself (recirculated) using `generate`. Lucid also supports two further ways to determine an event's destination in the network.

The first of these allows programs to send an event to a specific switch, regardless of its location in the network, by supplying a **switch id** when the event is generated. This can be done with the statement `generate_switch (id, event)`. Events sent via this method will be sent as regular traffic packets and delivered via the underlying network's routing algorithms, and so inherit the properties of those algorithms. Switch ids are typically not IP addresses; rather, they use a precomputed mapping from integers to switches that must be supplied to the Lucid compiler.

The second generation type allows users to send events out of multiple ports by utilizing the switch's multicast engine. Doing so requires the user to provide a **multicast group**, which is simply a set of ports. Multicast groups have type `group`, and can be defined in two ways:

1. The expression `{0, 3, 7}` specifies a group containing ports 0, 3, and 7. The entries of the group must be integer constants.

2. The expression `flood x` specifies a multicast group containing all ports *except* `x`. Unlike the prior syntax, `x` may be computed dynamically.

The `flood` syntax provides support for the common use-case of broadcasting a message to all ports but one (typically the port the message originally came from). The reason the first syntax requires all entries to be constant is because the switch's multicast engine must be configured ahead of time, so we must be able to statically determine all possible multicast groups in the program during compilation. We sup-

port dynamic computation of `flood` by predefining groups for every possible value of `flood x`, thus ensuring that the appropriate group always exists.

In either case, events may be generated with the statement `generate_ports (grp, event)`, which will result in `event` being sent out of all ports in `grp`.

3.2.5 Sizes

Memory is at a premium on PISA architectures, both for local variables and globals. Accordingly, dataplane programmers frequently wish to ensure their data takes up exactly the necessary amount of space and no more. Lucid enables this through the use of **sizes**, which are positive, compile-time integer values that represent bitwidths and vector lengths.

Sizes may be passed as parameters to certain types. The built-in types that accept sizes are listed below. User-defined types may also take size parameters, which can be passed through to any component types with a size parameter.

- Integers: `int<sz>` is the type of `sz`-bit integers. The type `int` is syntactic sugar for `int<32>`.
- Arrays: `Array.t<sz>` and `PairArray.t<sz>` are the types of (pair) arrays whose elements have type `int<sz>`. The length of the array is not part of the type. Pair arrays are discussed in §3.2.6.
- Vectors: `ty[sz]` is the type of vectors of length `sz`, whose elements have type `ty`.

In addition to constant sizes, users may declare size variables at top level, as follows:

```
1 size sz1 = 16;
2 extern size sz2; // Value must be supplied at compile time
3 size sz3 = sz1 + sz2;
```

In practice, we have found that size variables are typically used to denote size values that are either conceptually distinct, or represent parameters of the program that might vary be tweaked during later compilations. Accordingly, Lucid assumes that size variables are meant to be distinct, and therefore always treats a size variable as different from every other size, even if they "obviously" have the same value. Using the above example, the types `int<16>` and `int<sz1>` would be considered different types.

Sizes are not interchangeable with regular integers; the positions where sizes and integers may appear are disjoint. It is possible to cast a size to an int using the builtin `size_to_int` operator. Casting the other direction is impossible, since integers are runtime values and sizes must be determined at compile time. Besides casting, the only operator that may be used on sizes is adding two sizes together.

3.2.6 Advanced Array Accesses

Lucid's ordering restriction on global variable accesses means that a program cannot, in a single pass through the pipeline, read from an array, perform several computations, and write back to the same array. If a program *must* employ this logic, it can utilize recirculation, generating another event that will appear at the beginning of the pipeline. This is undesirable, because recirculation comes at a hefty performance cost (an entire new packet to process). Fortunately, there is an alternative: if the amount of computation is very small, the Tofino hardware provides the ability to do all three steps (read, compute, write) in a single array access.

Lucid models this ability using a construct called a **memop** (**Mem**ory **Op**eration). Memops are a special kind of function, which are syntactically restricted to ensure they can be performed as part of a single array access[3]. There are two kinds of memops, distinguished by the number of arguments they take. Basic 2-argument memops

[3]Specifically, memops are designed to fit in the processing capabilities of a single stateful ALU on the Intel Tofino.

are simple and useful in a wide variety of situations, while the powerful 3- and 4-argument memops are more complicated, but able to exploit the full capabilities of the Tofino's stateful ALUs.

Simple Memops Every memop is a function that takes two or more arguments. The first argument(s) represent the value in the array; the remaining argument(s) are supplied by the handler whenever the memop is used. The simplest kind of memop takes two arguments: the value in memory and one local value. These memops have heavily restricted syntax; their body may take one of two forms, as shown below.

```
1 memop just_return(int<'a> memval, int<'a> localval) {
2   return <e>;
3 }
4
5 memop conditional_return(int<'a> memval, int<'a> localval) {
6   if (<e>) { return <e>; } else { return <e>; }
7 }
```

More formally, a 2-argument memop's body must be either a single return statement, or a single `if` statement, both of whose branches are single return statements.

There are additional restrictions on the expressions `<e>` that appear in the memop. Each argument to the memop can appear at most once in each expression; furthermore, the only allowed expressions are variables, constants, and the following operations: `+, -, &, |, =, !=, <, >, &&, ||, !`.

The entire body of the memop must typecheck normally; e.g. the condition being tested must be a boolean. The return type of the memop is expected to be the same as the type of its first argument (i.e. some size of integer).

Usage In addition to `Array.get` and `Array.set`, the `Array` library provides accessor functions that incorporate simple memops. These are `Array.getm`, `Array.setm`, and `Array.update`. The first two are similar to `Array.get` and `Array.set`, but apply

a memop before returning the value/writing the value to memory. `Array.update` lets a user do both in a single action, potentially using different memops for each. As an example, consider the following code:

```
1  memop incr(int<'a> memval, int<'a> localval) {
2      return memval + localval;
3  }
4
5  memop max(int<'a> memval, int<'a> localval) {
6      if(memval > localval) { return memval; } else {return localval; }
7    }
8
9  event foo(...) {
10     // Sets x to the value at index, plus localval
11     int x = Array.getm(arr1, index, incr, localval);
12     // Sets the value at index2 to the maximum of itself and localval2
13     Array.setm(arr2, index2, max, localval2);
14     // Performs both operations simultaneously using the value at idx3
15     int y = Array.update(arr3, index3, incr, localval, max, localval2);
16 }
```

Each function takes at least one memop, which is followed by the local variable to be used as the second argument[4]. The call to `Array.getm` will retrieve the value at `index`, add `localval` to it, and store the result in `x`. The call to `Array.setm` will set the value at `index2` to the larger of the current value and `localval2`. Finally, the call to `Array.update` will perform both operations simultaneously (using the same index). Note that each memop is applied to the original value in memory, so the value of `y` depends only on `incr` and `localval`, not `max` or `localval2`.

Looking at the above example, one might notice that the usage of `Array.getm` seems overcomplicated; it would be equivalent to simply use `Array.get` and then

[4]In a functional language, this would typically be handled by partial application. Since Lucid is not functional, we pass the second memop argument to the `Array` function instead.

add `localval` to the result afterwards. However, on the Tofino, retrieving `x` and then incrementing it takes two stages of the pipeline, since the add must to take place *after* the read. In contrast, the computation performed during a memop is done in the same stage as the read. Judicious use of memops can be crucial to convincing a program to fit within the limited resources of a switch[5].

Pair Arrays and 4-Argument Memops While 2-argument memops are simple and suffice for most purposes, they are not quite capable of fully exploiting the capabilities of the stateful ALU; that is, there are some workable applications that they cannot express without recirculation[6]. For those, Lucid provides a more powerful form of memop that expresses the entire computation, both reading and writing, in a single function.

The most general form of memop is a 4-argument memop, which is used to access a special kind of array called a **pair array**, which is essentially an array with two values per index. These are a native hardware feature that are represented in Lucid by the type `PairArray.t`. Despite storing two values per index, Pair Array lookups are only able to return a single value, and can can only be accessed by using a 4-argument memop.

Like simple memops, 4-argument memops have harsh syntactic restrictions on their body. A 4-argument memop must have the form shown in Figure 3.9, except that certain parts may be omitted as noted.

The first two arguments are the two values stored in the PairArray; the latter two arguments are local variables supplied at the time of access. Every 4-argument memop contains two *built-in* variables `cell1` and `cell2`; unlike the other variables,

[5]One could imagine a compiler pass that automatically performs this transformation; such a pass is not currently present in the Lucid compiler, but could be added in the future.

[6]For example, a cache that needs to maintain both cached values and the timestamp when they were inserted. Inserting new entires requires checking the timestamp, possibly replacing the cached value, and then updating the timestamp only if the value was replaced. Without complex memops, the timestamp must be accessed at two separate points (before and after replacing the value).

```
memop foo(int<'a> memval1, int<'a> memval2,
          int<'a> localval1, int<'a> localval2) {
  bool b1 = <e>; // May be omitted
  bool b2 = <e>; // May be omitted

  // May be omitted entirely, or just the else branch may be omitted
    if (<cond>) { cell1 = <e> } else
  { if (<cond>) { cell1 = <e> } }

  // May be omitted entirely, or just the else branch may be omitted
    if (<cond>) { cell2 = <e> } else
  { if (<cond>) { cell2 = <e> } }

  // No else branch is permitted. May be omitted, in which case a default
  // return value will be used (provided as an additional argument to the
  // PairArray.update function that calls the memop).
  if (<cond>) { return <return_exp> }
}
```

Figure 3.9: The form of a 4-argument memop.

these names are hardcoded and cannot be changed by the user. Semantically, these represent the two cells of the array, and are initialized to `memval1` and `memval2`, respectively. At the end of the memop, the value assigned to `cell1` will be stored in the first cell of the array, and similarly for `cell2`. The return value of the memop will be returned to the handler.

The types of expressions are more distinct in 4-argument memops as well. As before, no argument may appear in any expression more than once; furthermore each expression may use at most one of the `memvals`, and at most one of the `localvals`. Beyond that, the different types of expressions have the following restrictions:

- `<e>`: Same as in 2-argument memops.
- `<cond>`: Must be a boolean combination of `b1` and `b2`.
- `<return_exp>` Must be one of the variables `cell1`, `cell2`, `memval1`, or `memval2`. This is the only place where `cell1` and `cell2` may appear in an expression.

4-argument memops are used by calling the `PairArray.update` function, which has the following signature:

```
1 PairArray.update(arr, idx, op, arg1, arg2, default)
```

The function takes a 4-argument memop `op` and applies it to the values stored in the array at `idx`, using `arg1` and `arg2` as the local value arguments, and returning `default` if the memop does not execute a return statement.

3-Argument Memops The final kind of memop is almost identical to the 4-argument version, but slightly modified for use on regular arrays that only have one value per index. The only syntactic difference is that they lack the `memval2` argument. Since regular arrays have only one entry per index, only the value of `cell1` is written back to memory when the memop finishes. The `cell2` variable may be still be used in the return statement at the end of the memop; otherwise, its value is ignored.

Completeness By comparison with the existing documentation for the Tofino, we believe that 3- and 4-argument memops fully capture the capabilities of the Tofino's stateful ALUs in arithmetic mode. They can express any operation that can be performed in the process of a single register lookup.

3.2.7 Matches and Tables

In addition to `if` statements, Lucid supports a more sophisticated control-flow construct in the form of `match` statements. Unlike `if`s, `match`es can have more than two branches, and can test several conditions at once.

A match statement compares one or more integer expressions against an ordered list of **rules**, each of which has a corresponding list of **patterns** and a block of code called its **body**. The `match` statement compares the expressions to each of the rules in order, and executes the body of the first rule whose patterns match. The syntax for match statements is borrowed from OCaml:

```
match (exp1, exp2, ...) with
    | pat1,  pat2,  ... -> { <code> }
    | pat1', pat2', ... -> { <code> }
    | ...
```

The expressions (`exp1, exp2`, etc) are permitted to be any integer expressions. Each pattern may be one of the following:

- The wildcard pattern `_`, which always matches.
- An integer value, which matches if the corresponding expression has that value.
- An integer variable, which matches if the corresponding expression has the same value as the variable.
- A bitstring, in which some or all of the bits may be replaced by an asterisk, e.g. `0b1**1`. A bitstring matches if all the non-asterisk bits match, so `0b1**1` matches all 4-bit integers that begin and end with a 1.

Branches are compared in top-down order, so if multiple branches match, the highest one will be executed. If no branches match, the match executes no code. A user can always prevent this by inserting a "default" branch at the end consisting of only wildcard patterns.

Non-default patterns that involve wildcards (either the wildcard pattern or a bitstring with wildcards) are called **ternary** patterns. Having any ternary patterns makes a `match` statement significantly more expensive when compiled to the Tofino[7], and so should be avoided when possible.

Lucid Tables Match statements are a direct representation of PISA match-action tables, which have identical semantics (match against an ordered list of rules, execute the body of the first one that matches). However, they only fully capture one kind

[7]It means the statement must be compiled to a ternary match-action table rather than an exact one, and hence must be stored in the more expensive TCAM memory.

of table: **static** tables, whose rules never change during execution. PISA switches, including the Tofino, also support the more powerful **dynamic** tables, whose rules can be modified at runtime.

Dynamic tables are a key part of network programming, and are often used to represent changing network conditions (e.g. how to route packets, which ports are open in a firewall, etc.). The drawback is that updating these rules must be done through the control plane; there is no mechanism for adding or removing a rule from inside the switch's pipeline. This also means that updating rules is a very slow process, since the control plane is orders of magnitude slower than the dataplane. Still, dynamic tables are an irreplaceable part of certain networking algorithms.

Lucid provides a representation of dynamic tables in the form of **Lucid Tables** (note: to distinguish between Lucid's representation and the actual match-action tables, we will always use the full term Lucid Table to refer to the former. Like a `match` statement, a Lucid Table contains several rules, and can be called during a handler to match those rules and execute some code. *Un*like a `match` statement, the rules are not contained in the program body, but are inserted by the control plane while the program is running. Furthermore, the body of each rule is not arbitrary code, but rather a restricted function called an **action**.

Actions Actions are similar to memops in that they are specialized, syntactically restricted functions; however, they differ in their purpose and restrictions. Figure 3.10 shows the form of two simple actions that could be used to implement a lookup table.

Unlike other functions, actions have two sets of parameters. The first set is passed when the action is installed in the table as part of a rule; the second set is passed when the action executes (after its rule matches). The body of an action must be a single return statement matching its return type (either an integer or a record).

```
// Result of a table lookup.
// Invariant: if hit is false, then val is a dummy value
type result = {
  bool hit;
  int val;
}

// Failed lookup, return a default value
// that is determined when the action is called.
action result miss_acn()(int default) {
  return { hit = false; val = default };
}

// Successful lookup, return a real value (v)
// that is determined when the action is created.
action result hit_acn(int v)(int default) {
  return { hit = true; val = v };
}
```

Figure 3.10: Actions to be used in a dynamic lookup table

```
table_type lookup_table = {
  key_size: (32, 8) // Match on a pair of 32-bit and 8-bit expressions.
  arg_types: (int) // Match-time argument type(s) for this table's actions
  ret_type: result // Return type of this table's actions
}

global lookup_table tbl = table_create<lookup_table>((hit_acn, miss_acn),
                                                     1024,
                                                     miss_acn());
```

Figure 3.11: Declaring a table, using the actions defined in Figure 3.10.

In the case of Figure 3.10, the return type of an action indicates whether the lookup was successful or not (`hit`) and if so, what `value` was retrieved. If the lookup failed, a default value, provided at match time, is returned instead.

Declaring Lucid Tables Figure 3.11 shows how to create a Lucid Table. The table's type must be declared beforehand; this type consists of (1) the type of the keys that the rules will match against, (2) the match-time parameter type(s) to the table's actions, and (3) the return type of the table's actions.

Like arrays, Lucid Tables are located in a particular stage of the pipeline. This means they are subject to the ordering restriction, and hence are declared using the

`global` keyword. When a table is created, it takes three arguments:

1. The list of possible actions that will be entered into the table. Each of these actions must have the appropriate return type and match-time argument type. This information is required when configuring the table in P4[8].

2. The maximum number of entries in the table.

3. A default action to be performed if no rule matches. Note that since the action is being installed in the table, its install-time arguments (if any) must be supplied.

Accessing Lucid Tables Finally, Figure 3.12 shows how Lucid Tables can be used in a handler. To perform a lookup, one calls the inbuilt `table_match` command, which takes as arguments the table to match on, the keys to match against, and any match-time arguments to that table's actions. It then performs the match, and returns the result of the executed action.

In Figure 3.12, the handler then proceeds to either print the result, if the lookup was a hit, or insert a new value into the table. It does so using the `table_install` command to add a new rule. Since installing a rule can only be done by the control plane, the `table_install` rule is *asynchronous*: rather than modifying the table directly, it sends a message to the control plane requesting a particular rule be installed. As a result, `table_install` does *not* count as a global access for the purposes of the global ordering.

The syntax for a rule is shown on line 13; it consists of a pattern (in this case, the two keys `k1` and `k2`), followed by the action to be executed when that pattern is matched. Again, we must supply the install-time argument(s) to this action at this point.

[8]One might wonder if we could determine the set of possible actions via static analysis, by examining every install command. Unfortunately, it may be the case that the control plane wishes to install rules that are never installed from the Lucid program. The best we can do is to ensure that all rules that *are* installed by the Lucid program use appropriate actions.

```
event lookup(int<32> k1, int<8> k2, int v) {
  int default = 0;
  // Global access: consumes the table
  result tbl_result = table_match(tbl, (k1, k2), (default));

  if(tbl_result#hit) {
    printf("Lookup successful, value %d", tbl_result#val);
  } else {
    printf("Lookup failed, inserting value %d", v);

    // Asynchronous operation: does NOT consume the table
    table_install(tbl,
      { (k1, k2) -> hit_acn(v); }
    );
  }
}
```

Figure 3.12: An event that performs a lookup in `tbl`, and either prints the result or adds a new entry for the current keys.

Patterns Like `match` statements, Lucid Tables support both exact matches (as in Figure 3.12) and ternary matches. Ternary patterns are entered into the table by means of a bitmask. The syntax for doing so is as follows:

```
int<8> mask = 0b11111000;
table_install(tbl,
  { (k1, k2 &&& mask) -> hit_acn(v); }
);
```

In the example above, the bitmask specifies that pattern will perform an exact match against `k1`, but will only match on the first 5 bits of `k2` (whose corresponding mask bits are 1s). The remaining bits of `k2` will be treated as wildcards (because their mask bits are 0s).

Priorities Also like `match` statements, Lucid Tables match their rules in a particular order, determined by each rule's **priority**. Priorities are integer values, with lower values being matched against first. By default, rules are entered into a table with a priority of 10; however, users can specify a different priority as follows:

```
table_install(tbl,
  // Install the rule with a priority value of 9
  { [9] (k1, k2) -> hit_acn(v); }
);
```

The above program installs the rule with a priority of 9, meaning it will be matched before any rules with the default priority of 10. If multiple rules have the same priority, they are matched in insertion order (i.e., new rules are matched last).

3.2.8 More on Types

By this point we have seen all the major types in Lucid. This section covered related topics that have not come up, or have not been covered in detail.

User-defined types As demonstrated briefly in the previous section, users can declare their own types, using the syntax

```
type foo<'sz1, 'sz2, ...> = ...;
```

where `'sz1, 'sz2, ...` are polymorphic size parameters that may be used in the body.

Notably, record types *must* be defined this way before they are used. Furthermore, the labels in a record type must be unique; this allows type inference to easily determine the type of a record from a single label.

Polymorphism Beyond type definitions, Lucid allows functions to be polymorphic in the type and sizes of their input arguments. Like before, polymorphic arguments are denoted by a name preceded by a tick. For example, the following function accepts any type for its first argument, and any size integer for its second:

```
fun void foo('a arg1, int<'b> arg2) { ... }
```

Global types A global type is one that represents persistent, mutable memory. This means each instance of a global type is located in a particular pipeline stage, and hence are part of the global order and must obey the ordering restriction (described in §3.1.3). Instances of global types must always be declared using a `global` declaration.

Formally, global types are defined by the following rules:

1. The types `Array.t` and `PairArray.t` are global.
2. All `table_type`s are global.
3. Compound types (records, vectors) are global if any of their components are global.

Constraints Functions that take global-typed arguments additionally carry **constraints** that describe what the relative order of those arguments must be; this is to ensure that global accesses within the function do not violate the ordering restriction. Constraints are specified following the arguments of the function, as below:

```
1 fun void foo(Array.t<16> arr1, Array.t<32> arr2, Array.t<1> arr3)
2              [start < arr1, arr1 < arr2 < arr3, end arr3]
```

This function has three constraints. The first specifies that the function must start (i.e. be called) before `arr1` has been used. The second specifies that `arr1, arr2`, and `arr3` must have been declared in that order, and the third specifies that when the function is done executing, `arr3` will have been accessed, and so cannot be accessed later (and neither can `arr1` and `arr2`, since they were declared before it).

In practice, we are always able to infer these constraints from the function body, so the user need not write them explicitly. They may still do so as a reminder, however, or if they wish to add artificial constraints on the function's usage. The one exception is that functions that appear in the interface of a module must be annotated with their constraints, since the interface does not provide the body of the function.

Events that involve global arguments also require constraints, and unlike functions these constraints cannot be inferred and must be supplied by the programmer. Since their starting and ending locations are fixed[9], event constraints only use the form `a < b`. Explicit annotations are required because every event can generate every other event, so events are essentially all mutually recursive; this severely complicates constraint inference. Fortunately, events that use global arguments are rare in practice, so this is not especially burdensome to the programmer.

3.2.9 Interface files

Lucid programs need not operate on their own. Indeed, they are capable of interoperating both with the control plane and with other Lucid programs, provided that all sides of the operation agree on the same *interface*. Lucid enables this agreement in two ways.

Interoperation with other Lucid programs is provided through **header files**, which declare certain events that should be common to both programs (although they may have different implementations). Programs can then include those files to ensure they have compatible interfaces.

Interoperation with the control plane is enabled via the use of **interface files**, which are generated by the Lucid compiler. There are two types of interface files: **event files** contain information about the events that are declared in the Lucid program, while **global files** allow the control plane to map the names of global values in the source program to their realization on the hardware.

Header files Header files in Lucid are used in the same way as languages like C++: by convention, they contain declarations but not definitions. The most important declarations in a header file are those of events, since events are how different switches

[9]The beginning and end of the pipeline, respectively.

communicate.

When events are compiled to the hardware, their names are replaced with integer identifiers that are used to distinguish packets of different event types. Since Lucid cannot guarantee that the compiler will choose the same identifier for two events in different programs (even if they have the same declaration), we allow users to select the identifier when an event is declared, as shown below:

```
1 event foo@1(int x);
2 event bar@2(int<16> y, int<8> z);
```

The above program defines two events named `foo` and `bar`, whose assigned identifiers are 1 and 2, respectively. Any program that includes these lines must assign those identifiers to the respective events, ensuring compatibility across programs.

Event files An event file is a Python program that has two parts: first, it contains a list of all the events that are declared in the Lucid program, including their name, arguments, and integer identifiers. Second, it contains Python functions for parsing and deparsing events from/to bytestreams. This file is automatically generated by the Lucid compiler, and can be used as the basis of a control plane program as it handles the sending and recieving of events.

Global files A global file is a JSON file that contains, for every global declared in the source program, the name of the hardware entity it corresponds to. This allows the control plane to know, for example, precisely how it should insert a rule into a given table, or which memory cell it should poll to obtain a count. Global files are also generated automatically, and are always needed when Lucid Tables are used.

3.2.10 Parsers

When an event arrives at a switch, it does so as a packet, in the form of an unstructured series of bits. These bits must be parsed to determine which event they correspond to, and what its arguments are. Conversely, when an event is generated, it must be deparsed (or "serialized") into a packet for transmission. For background events, Lucid generates an appropriate P4 parser and/or deparser automatically. However, parsing packet events (which correspond to traffic packets) requires knowledge about both the format of packets in the network, as well as the user's high-level intentions for what each packet event should represent.

Lucid allows users to express this information in the form of **parsers**. Parsers are special blocks of code that are capable of reading bits from a packet, matching on the result, and ultimately generating an event.

```
1 packet event
2           eth(int<48> dst, int<48> src, int<16> etherty, Payload.t payload);
3
4 parser parse_eth() {
5   read int<48> d;
6   read int<48> s;
7   read int<16> ety;
8   generate eth(d, s, ety, Payload.parse())
9 }
```

Figure 3.13: A very basic parser that parses the first three elements of an ethernet header

A simple parser is shown in Figure 3.13. When this parser is called on an arriving packet, it will read the first 48 bits into the local variable `d`, the next 48 bits into `s`, and the next `16` bits into `ety`. It will then use that data to create an `eth` event to be handled by the Lucid program, whose arguments are the parsed data, and whose payload is the rest of the packet (denoted by the call to `Payload.parse`).

A more sophisticated parsing setup is shown in Figure 3.14. It begins by defining record types for the two header types it cares about: ethernet and IPv4, as well as two events: one for a generic ethernet packet, and one for an IPv4 packet encapsulated

```
const int IPV4_ETY = 0x0800
const int IPV6_ETY = 0x86DD

type eth_hdr  = {dst : int<48>; src : int<48>; etherty : int<16>; }
type ipv4_hdr = { ... }

packet event eth   (eth_hdr e, Payload.t payload);
packet event eth_ip(eth_hdr e, ipv4_hdr ip, Payload.t payload);

parser parse_ip(eth_hdr e) {
    read ipv4_hdr ip;
    generate eth_ip(e, ip, Payload.parse());
}

parser main() {
  read eth_hdr e;
  match e#etherty with
  | LUCID_ETHERTY -> { do_lucid_parsing(e); }
  | IPV4_ETHERTY  -> { parse_ip(e); }
  | IPV6_ETHERTY  -> { drop; }
  | _ ->             { generate eth(e, Payload.parse()); }
}
```

Figure 3.14: A pair of more sophisticated parsers that parse IPv4 packets separately from other ethernet packets, and drop IPv6 packets.

in an ethernet packet. It then defines two separate (but related) parsers: one for parsing an ethernet packet from scratch (`main`) and one for parsing an encapsulated IPv4 packet (`parse_ip`). Lucid programs always begin parsing a new packet by calling the parser named `main`.

In this case the `main` parser begins by reading the ethernet header from the packet[10]. Reading a compound type in a parser is done by simply reading each of its components in declaration order. It then matches on the `ethertype` field of the just-parsed header to determine what action to take next.

If the ethertype contains the special value indicating that this is a packet generated by Lucid (i.e. a background event), then we transfer control to Lucid's automatically generated parser by called the built-in parser `do_lucid_parsing`. Otherwise, if this is an IPv4 packet, we continue parsing by calling the `parse_ip` parser. If this is an

[10]This parser encodes an assumption that all packets in the network are ethernet packets. An assumption of this form is always necessary to begin parsing at all. Packets violating the assumption are unexpected traffic in the network, and will produce unexpected behavior.

IPv6 packet, we drop it, and for any other type of packet we generate an `eth` event and finish parsing.

In the case of an IPv4 packet, the `parse_ip` parser reads the IPv4 header and generates an `eth_ip` event using both it and the previously parsed ethernet header. Instead of defining a separate parser for IPv4 packets, we could also have inlined the body of `parse_ip` into the `IPV4_ETHERTY` branch of `main`. Indeed, parsers can be nested arbitrarily deep in this way. However, defining separate parsers can be useful for modularity and code reuse (e.g. `parse_ip` could be called from different parsers, or different branches of the same parser).

Checksums Some header types, such as IPv4, include checksums to protect against data corruption during transit. Accordingly, Lucid provides a way of computing checksums during parsing, as shown below:

```
1 read ip_t ip;
2 int<16> new_csum = hash<16>(checksum, ip#v_ihl, ip#tos, ip#len, ip#id,
3                              ip#flags, ip#ttl, ip#proto, ip#src, ip#dst);
```

This hash function uses the built-in `checksum` seed to indicate that it should use the hardware's underlying checksum computation. The output is a hash of the provided fields, which can be used to update a checksum when a field is changed, or compared against the previous checksum to ensure that the packet is uncorrupted.

Automatically generated parsers Often, a Lucid program will only have one packet event, and the arguments to that event correspond directly to the header fields of the packet. This was the case for our very first example program (Figure 3.1), which had the same event declaration as Figure 3.13. One might notice that our argument to the `eth` event precisely describe the format of an ethernet header[11]: 48

[11] Although there may be variations depending on which version of the ethernet specification is used.

bits for the destination address, followed by 48 bits for the source address, then 16 bits for the ethertype, and then the rest of the packet (the payload).

When this happens (exactly one packet event whose arguments correspond directly to the packet format), it is possible to generate a parser automatically that simply associates all (non-Lucid-generated) packets with that event, and reads their arguments directly from the headers, in order. This is exactly the behavior of the parser defined in Figure 3.13. As a convenience, Lucid does precisely this whenever a program with one packet event does not define a `main` parser. For such programs, all parsers may be automatically generated, and users need not write their own at all.

Deparsers The deparser for Lucid events serializes them back into a series of bytes. Deparsers simply emit the packet headers as a stream of bytes in order; for background events, the header values are preceded by an ethernet header with a special Lucid ethertype. For background events, deparsers are always the inverse of parsers; however, since user-generated parsers may have arbitrary behavior, the deparser for a packet event may not be its inverse.

Comparison to P4 parsers Parsers in P4 [59] are represented as state machines, with a designated start state and two implicit `accept` and `reject` states. Each state may perform several operations such as reading from the packet or skipping forward, and ends with a transition statement indicating the next state.

There are two kinds of transition statements. Unconditional transitions simply specify the next state. Otherwise, the `select` keyword is used to match on one or more provided header fields to determine the next state. The semantics of `select` are the same as those of a match statement.

When designing the Lucid parser syntax, the primary goal was something simple and easy-to-use. As a result, Lucid parsers represent a subset of the capabilities of P4 parsers. The most notable distinctions are that P4 parsers may contain loops, while

Lucid parsers may not. Furthermore, P4 parsers may allow dynamically changing the patterns in their `select` statements, while Lucid parsers match against static patterns.

In practice, we have not found these to be significant limitation. PISA parsers do not support loops, so any parser loops in P4 would have to be unrolled anyway. Furthermore, dynamic patterns are needed for only a handful of protocols (such as MPLS), and so disallowing them does not severely hurt usability.

Header Slots Much like the global ordering, Lucid parsers must satisfy certain restrictions to ensure they can be compiled to hardware. When a Lucid program is compiled to P4, each local variable and event argument is stored in the header fields of the packet being processed. In particular, each argument is stored in a specific "slot"[12] in the packet's headers. During parsing, we need to ensure that when we read bits from the packet, they are stored in the appropriate slot, so they can be accessed while handing the event[13].

This means that we must determine which slot every `read` statement targets, which we can do easily via a static analysis. For example, in Figure 3.14, `e` should be read into the slot of `eth`'s first argument (which is also the slot of `eth_ip`'s first argument), while `ip` should be read into the slot of `eth_ip`'s second argument.

However, problems may arise if two variables or arguments need to share a slot. For example, consider Figure 3.15a. Variables `e1` and `e2` are both used as the first argument to `foo` at different points. This means they must both be read into the same slot, which is impossible, since they are both alive at the same time. The parser is therefore not implementable (as written) in P4, although in this case the issue could be solved by moving the reads for `e1` and `e2` into their respective branches, assuming they are not needed for the match.

[12] Analogous to a register.

[13] It would also be possible to move them into the appropriate slot after parsing, in the pipeline itself, but doing so would take potentially many pipeline stages.

```
1 read eth e1;
2 read eth e2;
3 match ... with
4 | 0 -> { generate foo(e1); }
5 | 1 -> { generate foo(e2); }
```

(a)

```
1 read eth e1;
2 match ... with
3 | 0 -> { generate bar(e1, ...); }
4 | 1 -> { generate bar(..., e1); }
```

(b)

```
1  match ... with
2  | 0 -> {
3    read eth e1;
4    match ... with
5    | 1 -> { generate foo(e1); }
6    | 2 -> { generate bar(e1, ...); }
7  }
8  | 3 -> {
9    read eth e2;
10   match ... with
11   | 4 -> { generate foo(e2); }
12   | 5 -> { generate bar(..., e2); }
13 }
```

(c)

Figure 3.15: Ways in which a parser could go wrong: (a) Two variables need to share the same slot at the same time, or (b)/(c) Two parameters to the same event need to share a slot.

The other problem that can occur is unifying different slots of the same event, as in Figure 3.15b. In this case, the same variable (`e1`) is used as both the first and last argument to `bar`, so those parameters must share a slot. Like before, this is impossible, since the parameters are alive at the same time. Problems of this type can even occur transitively. In Figure 3.15c, we experience the same issue, but more drawn out: lines 5 and 6 imply that the first parameters of `foo` and `bar` must share a slot, while lines 11 and 12 imply that the last parameter of `bar` must share the same slot[14].

Detecting slot overlaps Fortunately, Lucid is able to detect these sorts of errors and provide users with useful feedback when they occur. It does so using a unification-based analysis, much like the type system described in §4, inspired by algorithm J [53].

[14]In the future, we may be able to avoid this particular issue by using the P4 `lookahead` function to read `e1` multiple times into different slots.

We assign a fresh "slot variable" to each event parameter, parser parameter, and local parser variable. When a variable is used as an argument to an event or parser, we unify that variable's slot with the slot of the corresponding parameter. For example, in Figure 3.15b, line 3 unifies `e1`'s slot with the first parameter of `bar`, and line 4 unifies it with the last parameter of `bar`.

The sets of variables and parameters that have been unified together form an equivalence class representing a single slot: every element of that class must go in the same slot. We validate these equivalence classes by ensuring that no class contains:

- Two parser variables that are alive at the same time
- A variable and a parameter from the same parser, or
- Two parameters of the same event

If any of these occur in an equivalence class, we have detected an illegal overlap, and can report it to the user. In addition to reporting which things collided, we can also print the equivalence class itself, to give the user full information about *why* those two things were unified. This allows the user to more easily figure out what went wrong and how to fix it.

3.3 The Lucid Interpreter

By default, Lucid provides two tools for actually executing a Lucid program. The most obvious is the compiler: Lucid is compiled into P4, which can then be compiled to the Tofino and executed on actual hardware. This process is covered in-depth in Chapter 5.

However, Lucid also provides an interpreter, which can be used to simulate the behavior of a Lucid program without compiling and running it. Anecdotally, network programmers who used Lucid have commented on how much of an improvement this

is over P4 and similar languages; it is invaluable to be able to see how a program should behave without committing to the lengthy compilation/deployment process, or relying on the ponderous[15] ASIC simulator.

3.3.1 Simulation

The Lucid interpreter is somewhat more complex than a standard language interpreter. Rather than running a program line-by-line, it simulates a network of switches responding to and generating events. The topology of this network is determined by a configuration file which is supplied when the interpreter is invoked. Each switch runs a separate instance of the input Lucid program; in particular, note that this means that each switch has its own copy of every global variable in the program.

The interpreter maintains a global clock counting the number of steps since it began. By default, each step corresponds to roughly one nanosecond, but this is configurable. Each event in the network specifies the time and place it will appear. At each step of the interpreter, all events appearing at that time are processed in an arbitrary order. This might result in more events being generated, which will appear at a later time[16].

The interpreter prints each event to the console as it is processed, providing a log of execution. Additionally, users may use the builtin `printf` statements to print additional information, usually for debugging purposes (`printf` statements are ignored when running on actual hardware). At the end of interpretation, the interpreter also prints the internal state of each switch, so the user can directly inspect the final state of the network. Since termination is not guaranteed in general, users can set a maximum global time.

Users can interact with the interpreter while it is running, allowing them to gen-

[15]The ASIC simulator can simultate a switch processing about one packet per second.

[16]The time for an event to appear after being generated is configurable, and may be randomized to simulate network jitter.

erate events and receive events that were sent outside the network (e.g. a forwarded packet). This latter capability allows the interpreter to work as a softswitch. In particular, a user could use the interpreter to easily test Lucid programs in other simulation environments, e.g. Mininet [48] – this could be desirable if the other simulator has features the Lucid interpreter lacks.

The Lucid interpreter also has a non-interactive mode, where the entire trace of input events is provided at the start. This is useful for testing the behavior of a program on a large trace, such as actual traffic data. Indeed, Chapter 6 discusses how we use this capability to automatically optimize Lucid programs by testing various configurations against captured traces using the interpreter.

3.3.2 Capabilities of the Interpreter

What does the interpreter model? In additional to simply interpreting Lucid programs, the interpreter provides a basic model of several features of real networks. In particular, the interpreter models:

- **Multi-switch topologies**: The interpreter lets users simulate a program in a setting with multiple switches that each have independent connections and memory; both the number of switches and the topology are configurable by the user.

- **Time**: One step of the interpreter corresponds to one nanosecond of actual time; this is a good approximation of modern packet processing speeds. Furthermore, operations such as event generation have built-in delays, both for creating the event in the first place and for sending it across a link, if necessary. These values are configurable by the user.

- **Packet Reordering**: In actual networks, packets may unpredictably arrive or be processed in a different order than they were sent. Lucid models this by

adding a random delay to events when they are generated or sent across a link. The range of delay values is configured by the user, and random delays can be disabled by setting this range to 0. Furthermore, a user may provide a random seed to ensure determinism.

- **Packet Loss**: Similarly to reordering, packets may be lost or corrupted when transmitted across links. Lucid models this by providing a user-configurable chance of packets being randomly dropped each time they are sent across a link.

- **The Control Plane**: Lucid supports control-plane programs written in Python which can send and receive events, as well as arbitrarily modify switches' memory during interpretation. If a control plane file is provided, an instance of the Python interpreter is maintained alongside the Lucid interpreter, with pipes between them for communication.

What doesn't the interpreter model? There are several important features of real networks that are not covered by the interpreter. Typically, this is because the interpreter operates within the framework of Lucid, and the feature in question is not represented in the language.

- **Resources**: Lucid's interpreter does not model the finite resource constraints of a real switch, since those constraints are not reflected in the surface language[17]. The interpreter will happily run a program that is far too large to fit on a real switch.

- **Queuing**: Events in the interpreter are stored in a single, undifferentiated queue with no length limit. Real switches typically have multiple queues (e.g.

[17]It does model ordering constraints, but this model is moot since ordering errors will be caught by the type system before interpretation.

to sort packets by priority), of finite length. The interpreter will never drop an event because it did not have queue space.

- **Bandwidth**: Links have no limit on the number of packets that can be transmitted across them in any given timeframe.

3.4 Evaluation

The Lucid project has one primary goal: make it easier to write dataplane programs. We address this primarily through language design: we have endeavored to make the syntax intuitive and easy to reason about, and to provide high-level abstractions over complicated switch features, such as events to represent recirculation and handlers instead of chains of match-action tables. In particular, we have created high-level representations of low-level hardware restrictions, such as the global order and memops, and combined them with easy-to-understand guidelines and feedback to ensure those restrictions are enforced.

Accordingly, to evaluate Lucid, we should ask the following two questions:

1. **Expressivity.** Can Lucid be used to express a wide variety of useful and practical dataplane programs, despite being a higher-level, more abstract language?

2. **Usability.** Is Lucid significantly easier to use than existing dataplane languages?

3.4.1 Expressivity

In our initial evaluation of Lucid, we implemented a benchmark suite of 10 dataplane applications, listed in Figure 3.16. These applications range from simple NATs to complicated analytic programs like *Flow [74]. We were able to express the functionality of each program entirely in the dataplane, and always with much less effort than

Application	Description	LoC Lucid	LoC P4
Stateful Firewall (SFW)	Blocks connections not initiated by trusted hosts. ***Control events update a Cuckoo hash table.***	189	2267
Fast Rerouter (RR)	Forwards packets, identifies failures, and routes. ***Control events perform fault detection and routing.***	115	899
Closed-loop DNS Defense (DNS)	Detects/blocks DNS reflection attack with sketches & Bloom filters. ***Control events age data structures.***	215	1874
*Flow [74]	Batches packet tuples by flow to accelerate analytics. ***Control events allocate memory.***	149	1927
Consistent Shared State (SRO)[86]	Strongly consistent distributed arrays. ***Control events synchronize writes.***	94	897
Distributed Prob. Firewall (DFW)	Distributed Bloom filter firewall. ***Control events sync. updates.***	66	1073
+Aging	***Adds control events for aging.***	119	1595
Single-dest. RIP	Routing with the classic Route Information Protocol (RIP). ***Control events distribute routes.***	81	764
Simple NAT	Basic network address translation. ***Control events buffer packets and install entries.***	41	707
Historical Prob. Queries (CM)	Measures flows with sketches for historical queries. ***Control events age and export state periodically.***	93	856

Figure 3.16: Applications with data plane-integrated control, implemented in Lucid and compiled to the Barefoot Tofino. The role of control events is bolded. Chart adapted from the original Lucid paper [73]

it would take to write them in P4. Each of these programs successfully compiled to the Tofino.

These examples provide a broad sample of the possible Lucid programs, but they are far from exhaustive. Later chapters of this thesis will discuss further programs we have written, and there have already been academic works using Lucid as a base, notably Parasol [37] (discussed in Chapter 6) and SwitchLog [52]. Lucid is being used by researchers outside of the Lucid developers, all of whom are creating more and more Lucid programs. It seems that Lucid is, in fact, capable of expressing interesting

Application	NAT	RIP	Dist FW	Dist FW + Aging
Dev. Time	25m	40m	25m	25m + 30m

Figure 3.17: Time for a student without Tofino experience to write Tofino-compiling Lucid applications.

dataplane programs.

3.4.2 Usability

We have several data points indicating that Lucid is easier to use than P4.

Program size As shown in Figure 3.16, Lucid programs are typically around 10x shorter than equivalent P4 programs. This measurement comes with an important caveat, however: with one exception, we are comparing Lucid code to the output of the Lucid compiler, not to hand-written P4. The reason is that, simply put, these applications are *so difficult to write* in P4 that we could not find any hand-written versions to compare against. The exception is *Flow, a complex application that did publish a P4 implementation. Fortunately, the comparison is in line with the trend: the Lucid version of *Flow was able to implement equivalent functionality in less than 1/10th the space.

Although the rest of comparisons in Figure 3.16 are in some sense artificial, the mere fact that we could not find handwritten implementations of these programs is a testament to the utility of Lucid.

Programming time We timed how long it took for one of the Lucid authors, a 3rd-year PhD student with no prior dataplane or Tofino programming experience, to implement several of the applications in Figure 3.16; the results are shown in Figure 3.17. Most applications were written in less than half an hour, with even the most complicated one (a distributed firewall whose entries time out) taking less than an hour.

```
1  apply {
2    if (hdr.ip.dstip == 1){
3      tmp = reg1_r.execute(0);
4      hdr.ip.dstip = reg2_rw.
       execute(0);
5    } else {
6      tmp = reg2_r.execute(0);
7      hdr.ip.dstip = reg1_rw.
       execute(0);
8    }
9  }
```

(a)

```
1  event packet_in(header_t hdr) {
2    if(hdr#ip#dstip == 1) {
3      int tmp = Array.get(reg1, 0);
4      Array.set(reg2, 0, tmp);
5    } else {
6      int tmp = Array.get(reg2, 0);
7      Array.set(reg1, 0, tmp);
8    }
9  }
```

(b)

```
error: Table placement cannot make any more progress. Though some tables have
not yet been placed, dependency analysis has found that no more tables are
placeable. This may be due to shared attachments on partly placed tables; may
be able to avoid the problem with @stage on those tables
```

(c)

```
In tmp/Errors/ordering_error2/ordering_error.dpt:

8|    Array.set(reg1, 0, tmp);
      ~~~~~~~~~~~~~~~~~~~~~~~

error: tmp/Errors/ordering_error2/ordering_error.dpt: Function call violates the
global order
```

(d)

Figure 3.18: A comparison of how the P4 and Lucid compilers respond to the same error. The P4 code and Lucid code are shown in (a) and (b), respectively. The P4 error is shown in (c) and the Lucid error in (d).

Certainly, the fact that the test subject was a Lucid author contributes to these short times, but as a point of comparison, the authors have generally found that it can take new PhD students several *weeks* of learning P4 and the Tofino before they can write something nontrivial. Even for an expert, this level of productivity in P4 is hard to imagine.

Error Messages Lucid provides users with guidelines for how to write correct programs, most notably via the global order and via the abstraction of memops. When those guidelines are violated, Lucid provides *useful, actionable* error messages pointing the users to the specific parts of the program that contain errors. Figure 3.18 compares the errors given by the P4 and Lucid compilers to the same fundamental problem (an ordering error – the two branches access `reg1` and `reg2` in different

```
1  RegisterAction<bit<32>,
2                 bit<8>,
3                 bit<32>>(reg1)
4    reg1_w = {
5      void apply(inout bit<32> remote,
6          out bit<32> ret_remote){
7          remote =
8            hdr.ip.srcip + hdr.ip.srcip
       ;
9          ret_remote = 0;
10     }
11   };
12
13   apply {
14     hdr.ip.srcip = reg1_w.execute(0);
15   }
```

(a)

```
1  memop double(int memval, int x)
2  {
3    return x + x;
4  }
5
6  ...
7
8  Array.setm(arr, 0, double,
9             hdr#ip#srcip);
```

(b)

```
error.bfa:71: error: Can't reference phv in first operand of add instruction
failed command assembler
```

(c)

```
In tmp/Errors/salu_error1/error.dpt:

7|    return x + x;
            ~~~~~

error: tmp/Errors/salu_error1/error.dpt: Second use of a local parameter in a
single memop expression
```

(d)

Figure 3.19: A comparison of how the P4 and Lucid compilers respond to the same error. The P4 code and Lucid code are shown in (a) and (b), respectively. The P4 error is shown in (c) and the Lucid error in (d).

orders).

The P4 snippet in Figure 3.18a results in the error in Figure 3.18c. Notice that the error does not describe the root cause of the issue (it just describes the particular problem that halted the compiler), nor does it indicate which part of the program is responsible. In contrast, the Lucid snippet in Figure 3.18b results in the error in Figure 3.18d, which both identifies the type of error (an ordering error) and points to the specific line of code where the error was detected.

Figure 3.19 performs the same comparison for memops. Figures 3.19a and 3.19b show the same program in P4 and Lucid, respectively. The P4 code results in the error in Figure 3.19c, while the Lucid code outputs the error in Figure 3.19d. Notice that

the P4 error references things (phvs, add operands) that do not appear in the code body. In contrast, the Lucid error message points to a particular line, and provides the specific error: one is not allowed to use a memop parameter (in this case `x`) more than once per expression[18].

Interpreter Lucid has an interpreter, which can be used to quickly and easily test the behavior of Lucid programs. P4 has a reference implementation, but it does not include the Tofino's architecture-specific externs. In practice, to "test" a P4-Tofino program, one must compile it to the hardware and then run it on Intel's proprietary software model of the Tofino. This workflow has a few problems: first, it requires that the program actually fit into the hardware before testing it; second, if the program does something unexpected, it may be due to a bug in either the program or the compiler; third, both the compiler and Tofino software model are slow. Lucid's interpreter solves all of these problems, and indeed our work on Parasol (described in Chatper 6) relies on the interpreter to perform its simulations.

Practical Impact Finally, several researchers at Princeton have begun to use Lucid in their work, either instead of or in addition to P4. These users have generally reported that Lucid is significantly easier than P4 to work with, and that they have experienced higher productivity and less frustration as a result. At the time of writing, Lucid has been used for at least the following projects[19]:

- Several undergraduate summer research projects, most notably SwitchLog [52], a variant of Datalog that runs in the dataplane.
- Parasol [37], a dataplane optimizer described in Chapter 6.
- Tango [12], a system for cooperative, performance-aware routing.

[18]The full details of valid memop syntax are described in §3.2.6, and are available to users at the Lucid GitHub repo.

[19]The first two projects involved the author; the remainder did not.

- SmartCookie [84], a SYN flooding defense system, which was prototyped in Lucid and later manually implemented in P4 to take fewer hardware resources.
- Orbweaver [85], a framework for exploiting unused bandwitch for in-network communication.
- A compact data structure for detecting out-of-order TCP packets [90]. An author commented that Lucid was chosen because it was "easier to learn [than P4]".

3.5 Comparison to P4

3.5.1 Feature Comparison

Although Lucid and P4 are both used to write dataplane programs, the structure of those programs is significantly different. Many Lucid features do not correspond one-to-one with P4 features; nonetheless, it is instructive to compare the ways in which the two languages address different hardware features.

1. **Target Architecture.** Lucid only supports programs that run on a Tofino switch. P4 is more versatile, supporting programs that run on multiple kinds of hardware through the use of **architecture files** that define various hardware-specific features. In the rest of this section, we compare Lucid specifically to the P4 language with the Tofino architecture files.
2. **Architectural Model.** Lucid supports programs that run in a single ingress and egress pipeline. P4 programs can utilize multiple pipelines, either in parallel to support additional ports, or serially to perform additional processing on each packet.

3. **Metadata.** P4 programs require extensive definitions of packet headers and metadata (e.g. the port on which a packet arrived or to which it is being sent), which must be explicitly managed in every program. Lucid programs only require the user to manage information which the program actually uses, in the form of event arguments. Full header definitions are only needed for writing custom parsers.

4. **Overarching Control.** Lucid programs are structured as a series of event and handler definitions, whereas P4 programs are structured as a series of **control blocks** describing the behavior of different hardware components (e.g. the ingress and egress pipelines). Lucid's compiler automatically interleaves different threads of control into a single P4 control block; in P4, this must be done manually as the program is written.

5. **Per-Packet Control.** Handler bodies in Lucid are written as imperative, C-like code. P4 control blocks may be written in the same way, but in practice they are structured as a series of match-action table definitions, followed by applications of those tables[20].

6. **Match-Action Tables.** P4 provides all four types of match-action table as language primitives, all of which must be defined separately from their uses. Lucid only uses this pattern for dynamic tables; static tables are provided via `match` statements, and generated automatically during the compilation process.

   (a) **Dynamic Tables.** Declarations of dynamic tables are essentially the same in both languages, but their uses differ. In Lucid, one "calls" a table, providing the keys and action arguments at that time. In P4, one "applies" the table, passing no information at all; the key and argument values must

[20]Code written this way will typically take fewer resources, and is much more likely to compile in the first place.

be set up in advance via assignments to designated variables. Furthermore, Lucid Tables return values when called, while P4 tables directly modify variables in the scope of their control block.

7. **Stateful Memory.** Lucid offers the uniform abstraction of arrays to represent the persistent memory available in PISA pipelines, and provides an ordered type system to ensure it is used in a consistent manner. P4 programs provide **registers**, which behave similarly, but may lead to compilation errors if used in an inconsistent way.

8. **Memory Accesses.** Both Lucid and P4 allow their stateful memory constructs to be accessed by special accessor functions: `memop`s in Lucid and `RegisterAction`s in P4. `memop`s are syntactically restricted to ensure they can be compiled to the hardware; `RegisterAction`s are essentially arbitrary blocks of code which may fail to compile for unexpected reasons.

9. **Packet Forwarding.** In Lucid, packets are emitted by generating the corresponding event at a particular location in the network (e.g. specifying the port to send the packet out of). In P4, packets are emitted by setting one of several metadata values to indicate the appropriate network location.

10. **Packet Delays.** Lucid provides the ability to buffer events to be sent out at a later time. P4 does not provide this natively; users would have to manually manage the Tofino's pausable packet queues (as Lucid does automatically).

11. **Parsers.** P4 parsers are structured as state machines that read from the packet and then match on its data to determine the next state. Lucid parsers are represented as nested match statements, a slightly more restrictive syntax – in particular, Lucid does not allow parser loops. This does not sacrifice completeness because the hardware does not allow loops anyway; the P4 compiler will

try to automatically unroll any loops during compilation. The parser operations are the same, albeit with different names (e.g. Lucid's `read` is equivalent to P4's `packet.extract`).

12. **Hash Units.** These have essentially the same behavior in Lucid and P4, but different interfaces. In P4, one must declare a `crc` object, which is passed to a `hasher` object, which can finally be called with the arguments to be hashed. In Lucid, one simply writes a single `hash` expression.

13. **Modules.** Lucid's system of modules is not present in P4.

14. **Functions.** Lucid permits standard C-style functions. P4 does not contain arbitrary user-defined functions, but does have several function-like primitives. The closest to a function is a **control block**, an object that defines some internal state that persists across packets. Each control block contains a single method named `apply`, which may have input and output arguments and has access to the variables defined in the block. The Tofino compiler imposes many restrictions on how control blocks can be used; for example, each control block can only be used once in each control flow path.

    (a) **Actions.** Control blocks may also contain **actions**. P4 actions are a type of **open function** that can be called from a control block's `apply` method or other actions. Open functions behave like standard functions except that, if they contain free variables, they use the definition of those variables at their call site, rather than their definition site. In contrast, actions in Lucid are standard functions with restricted syntax, which are called only by dynamic tables.

15. **Statements and Expressions.** The statements and expressions that appear within Lucid handlers are essentially the same as those in P4 control blocks,

except for the addition of vectors and loops to Lucid.

3.5.2 Limitations

Lucid compiles to P4; therefore, every Lucid program can be expressed as a P4 program. However, the reverse is not true: there are some programs that can be written in P4 but not Lucid. Most of the differences have to do with Lucid's assumption of a fixed architecture (a single ingress and egress pipeline on the Intel Tofino), and could likely be addressed by future work that expands the scope of Lucid.

Portability P4 can be used to program different kinds of switches, and even different hardware architectures entirely. However, programs written for one piece of hardware frequently end up relying on hardware-specific features, and thus fail to compile to any other targets. In order to help ensure that Lucid programs compile successfully, we made the choice to specialize the language to the Intel Tofino. Several of Lucid's features (most notably `memop`s) are targeted specifically at the capabilities of the Tofino. In the future, we hope to extend the range of targets Lucid can be compiled to; doing so will require work in figuring out how to add/extend restrictions to handle the various idosyncracies of different types of networking hardware.

Exploiting the Architecture Even on the Tofino, Lucid does not fully exploit every part of the architecture that can be programmed with P4. As mentioned earlier, Lucid assumes programs run in a single ingress and egress pipeline, but P4 programs can write programs that span multiple pipelines (at the cost of lower throughput). Furthermore, Lucid does not allow any configuration of the Traffic Manager (TM) that sits between ingress and egress pipelines. P4 programs can direct packets to specific queues in the TM, or add annotations to help classify traffic, which cannot be done in Lucid.

There are also several hardware features inside pipelines that can be used in P4 but not Lucid. Tofino stateful ALUs provide a "math mode" that can perform low-precision floating-point operations using predefined lookup tables. The Tofino also provides **meters**, a type of stateful object like arrays that are specialized for detecting if certain packet flows are exceeding a rate threshold.

Optimality Lucid's compiler[21] is not fully optimal with respect to hardware capabilities; that is, it may use more resources than necessary when fitting a program onto a switch. This is an unfortunate side effect of its abstraction relative to P4, which models hardware features more directly and thus gives users more control over resource allocation.

Relatedly, there are certain hardware features of the Tofino that are underutilized; for example, each stage of the pipeline contains special **gateway tables**, which can evaluate certain boolean conditions for "free" (i.e. without consuming a stage). Our compiler currently uses these only to check if Lucid Tables should be executed, but they could be used more broadly to precompute conditions without using stages.

3.6 Related Work

3.6.1 Other network programming languages

Over the past decade, researchers have developed a number of languages for network programming. For example, Frenetic [30] was designed to program OpenFlow controllers: Frenetic computations sat on a software server and generated lists of packet-processing rules to be sent to switches. These lists of packet-processing rules were described using their own domain-specific sublanguage. Over time, that sublanguage evolved and developed in work on NetCore [67], Pyretic [62], and NetKAT [7].

[21] For more details about the compiler, see Chapter 5.

Other languages, like FlowLog [56] and Maple [81] used other kinds of programming paradigms to control these OpenFlow systems at a high level of abstraction. A key distinction between earlier work based on OpenFlow, and later work based on P4, is that P4 switches contain persistent, mutable and programmable state. NetKAT (for example) is stateless and cannot describe or implement the stateful applications developed in this paper. The pipeline compilation and safety issues described in this paper do not arise in these more limited systems.

More recently, there have been a number of efforts to make programming P4 switches easier. For example, Domino [70], Chipmunk [32], Lyra [31], O4 [4] and P4All [36] allow programmers to use high-level, imperative, C-like languages to describe switch computations. They then typically deploy program synthesis techniques to allocate those computations to stages in the pipelines of one or more hardware devices.

When these techniques succeed in solving the synthesis problem, they are highly effective. However, these tools provide little or no feedback when their synthesis fails. We view Lucid's contributions to this space as complementary to, and synergistic with, these other efforts – one can certainly imagine future systems that adopt Lucid's high-level representations of hardware restrictions (its type system, in particular), and then use synthesis techniques with fewer possible reasons for failure. Indeed, Lucid's vectors and loops were inspired by related unsafe features in P4All [36]).

Abstraction strategies Most recent dataplane programming languages attempt to raise the level of abstraction in some way in comparison to P4. Different languages use different techniques. For example, Lyra [31] offers a "one big switch" abstraction, aimed at automatically distributing computation across several switches in the network while allowing users to program a single logical switch. Domino [70] and Chipmunk [32] provide a concept of **packet transactions** which capture the

logic for processing a single packet, as well as **atoms** that can be used to specify the instruction set of a particular router. P4All [36] extends P4 with "flexible" data structures, whose size is not hardcoded but can be determined as the result of an optimization.

A final, particularly notable point of comparison to Lucid is O4 [4], which aims to augment P4 with higher-level constructs, many of which have direct analogues in Lucid: arrays and loops (analogous to vectors and loops, respectively), as well as factories (very roughly analogous to modules). A key difference between O4 and Lucid is that O4 is an extension to P4, while Lucid is its own language.

Event-based programming To our knowledge, Lucid's adoption of events as the primary programming model is unique among dataplane languages (although it bears a resemblance to Domino's packet transactions). Event-driven programming is not new, of course, but was notably proposed for a networking setting in 2019, albeit as an extension to P4 rather than the basis of a language [39]. We believe the event-based model is a natural one for dataplane programming, and hope it will be adopted by other high-level dataplane languages in the future.

3.6.2 Syntax

One of the ways Lucid attempts to make dataplane programming easy is by providing intuitive syntax that makes programs easy to reason about. In many cases, the syntax is borrowed from other languages.

The intial concept of Lucid was to provide a simple, imperative syntax akin to C or C++. This can be seen in the structure of function bodies and event handlers, as well as the syntax for polymorphic arguments.

Much of Lucid's syntactic DNA is drawn from OCaml, including immutable values, polymorphic type variables (beginning with a tick), modules and interfaces, match

statements, and the unfication-based type inference system. In general, the authors drew heavily on OCaml both due to personal familiarity and a desire to imbue Lucid with a similar "functional style" of programming, in order to make it easier to reason about.

Finally, Lucid's syntax for creating lists via comprehensions is borrowed from Python.

3.6.3 Network Simulators

Lucid's interpreter can be used to simulate the behavior of a single switch or a network of switches. Traditionally, network programmers have used network simulators for this purpose. For Tofino programmers, the two primary methods were using Intel's proprietary ASIC simulator, which simulates in detail the behavior of a P4 program on the Tofino, or Mininet [48], a virtual network simulator.

In comparison to Lucid's, the ASIC simulator provides higher fidelity by modeling the internals of the switch processing a packet, but is up to *10,000X slower* and cannot model networks of multiple switches. In contrast, Mininet does model a whole virtual network, which can run the actual applications a network would use. Mininet has a higher-fidelity representation of a network than Lucid's interpreter (e.g. it models bandwidth). An interesting direction for future work would be integrating the two, using Lucid to model the individual switches and Mininet to model the network between them.

Chapter 4

Pipeline Types

One of the design goals of Lucid is to *make things easy*, and a primary enabler of that is its type system, which is designed to not only catch ordinary type errors, but also a class of **ordering errors** that can occur in switch programs. This chapter discusses how Lucid uses its system of **Pipeline Types** to not only catch ordering errors early in the compilation process, but to return *useful, actionable* feedback to the programmer to help them repair their code.

4.0.1 Attribution

The contents of this chapter were designed and implemented by the author, with advice from Dave Walker and John Sonchack. The text of this chapter is adapted from the POPL'22 paper describing this type system [51].

4.1 Pipeline Types by Example

4.1.1 Ordering Errors

Consider the simple Lucid program in Figure 4.1. It declares two arrays, `arr1` and `arr2`, and whenever it receives a `simple` event, it reads the value in `arr1[0]`,

```
1 global Array.t<32> arr1 = ...;
2 global Array.t<32> arr2 = ...;
3
4 handle simple() {
5  int x = Array.get(arr1, 0); // Read arr1[0], store in local x
6  int y = x + x;
7  Array.set(arr2, 0, y);      // Store y into arr2[0]
8 }
```

Figure 4.1: A simple Lucid program that copies a value from `arr1` to `arr2`, doubling it in the process.

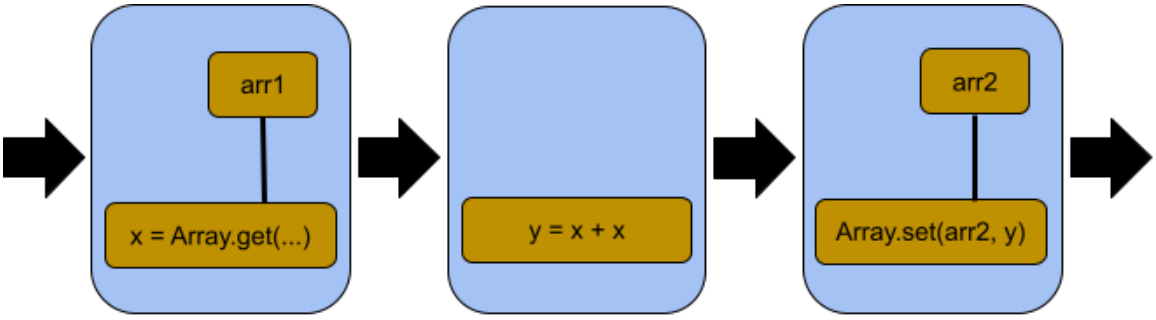

Figure 4.2: What the program in Figure 4.1 looks like when compiled to a 3-stage PISA pipeline.

doubles it, and stores the result in `arr2[0]`.

Recall from Chapter 2 that PISA switches process packets using a pipeline of stages, each of which performs some amount of work (its **actions**) and contains a small amount of persistent, mutable memory (**registers**). Figure 4.2 shows a simple 3-stage pipeline implementing the program in Figure 4.1. Arrays are stored in a stage's registers, which are isolated: they cannot be accessed outside of the stage that stores them. Hence actions that access an array must be located in the same stage as that array.

The process for compiling the code in Figure 4.1 to the pipeline in Figure 4.2 is straightforward. Each statement in the `simple` handler is a single action, so the dependencies between them determine the layout. The program must read `arr1` before computing `y`, so `arr1` and its read are placed into stage 1; symmetrically, `arr2` and its write are placed in after computing `y`, in stage 3.

```
global Array.t<32> arr1 = ...;
global Array.t<32> arr2 = ...;

handle simple() {
 int x = Array.get(arr1, 0); // Read arr1[0], store in local x
 int y = x + x;
 Array.set(arr2, 0, y);      // Store y into arr2[0]
}

handle simple_reversed() {
 int x = Array.get(arr2, 0); // Read arr2 first this time
 int y = x + x;
 Array.set(arr1, 0, y);      // Copy y into arr1 this time
}
```

Figure 4.3: A minor extension of Figure 4.1 that adds a new handler that accesses the arrays in the opposite order. Unfortunately, the program is now uncompilable!

Compiling Lucid programs is not always this simple. Imagine making a small extension to Figure 4.1 by adding a new event whose handler performs the same operation, but copies from `arr2` to `arr1`. Such an extension is shown in Figure 4.3.

Unfortunately, attempting to compile the new program runs into a problem: the `simple` handler wants to access `arr1` before `arr2`, and hence requires `arr1` to appear earlier than `arr2` in the pipeline. However, the `simple_reversed` handler requires the exact opposite! We cannot place `arr1` both before and after `arr2`[1]; therefore, it is impossible to compile this program such that both handlers can execute in a single pass through the pipeline. This is called an **ordering error**.

One way to resolve ordering errors is to use recirculation to send one of the events through the pipeline twice, accessing one array each time. However, doing so is very expensive, since it takes up bandwidth that could be used for processing other packets; furthermore, if the handler wished to access many arrays, the packet might require multiple recirculations, deepening the expense. As a result, rather than try to solve ordering errors automatically, Lucid instead opts to *detect* ordering errors, and provide users with *useful, source-level feedback* when they are detected.

[1]It is also impossible to place them in the same stage, since there are data dependencies between them.

4.1.2 Pipeline Types

To detect errors, Lucid uses a type-and-effect system named **Pipeline Types**, which is designed to enforce two constraints on the global values[2] in the program:

1. No global value is accessed twice in a single pass through the pipeline (since the packet moves to the next stage after accessing it)

2. There is some order on globals such that for every pair of accesses on each control path, the global accessed first appears earlier in the order.

To demonstrate the ideas underlying the type system, we will return to our running example of a Bloom filter from Chapter 3. We will again build up a Bloom filter program in several steps, each time adding more features and describing how to typecheck them. The basic version is replicated in Figure 4.4.

As Lucid typechecks the program in Figure 4.4, it tracks not only the *raw type* of each global (e.g. `Array.t<1>`), but also its *location*: an integer representing a stage in an abstract pipeline. The abstract pipeline has an infinite number of integer-indexed stages, each of which stores a single global variable. Globals are assigned to stages in declaration order, so in this case `a0` is stored in stage 0, while `a1` is stored in stage 1. Accordingly, we write the **full type** (or just "type") of `a0` as `Array.t<1>@0`, and similarly `a1` has type `Array.t<1>@1`. Thanks to Lucid's type inference, the programmer does not need to write the full type of globals; it suffices to write only the raw type, as in Figure 4.4.

As Lucid checks that a series of statements or expressions is well-formed, it keeps track of where the computation is in the abstract pipeline (the **current location**). Figure 4.5 demonstrates the process. Whenever a global is successfully accessed, the current location moves to the stage after that global, preventing it or anything before it from being accessed again. Statements that do not involve global accesses do not

[2]That is, arrays and Lucid Tables.

```
const int m = ...;
// Declare two 1-bit arrays with m entries each, initialized to 0
global Array.t<1> a0 = Array.create(m);
global Array.t<1> a1 = Array.create(m);
const int s0 = ...;  // Seed for first hash function
const int s1 = ...;  // Seed for second hash function

// Add item to filter
fun void add(int item) {
    int idx0 = hash(s0, item);
    int idx1 = hash(s1, item);
    Array.set(a0, idx0, 1);
    Array.set(a1, idx1, 1);
}

// Return true if item in filter
fun bool query(int item) {
    int idx0 = hash(s0, item);
    int idx1 = hash(s1, item);
    int<1> b0 = Array.get(a0, idx0);
    int<1> b1 = Array.get(a1, idx1);
    return (b0 == 1 and b1 == 1);
}
```

Figure 4.4: A basic Bloom filter with k = 2. Functions `add` and `query` may be called from many different handlers.

```
global Array.t<1> a0 = Array.create(m); // Stored at location 0
global Array.t<1> a1 = Array.create(m); // Stored at location 1

fun void add(int item) {       // Starting location: 0
    int idx0 = hash(s0, item); // Current location : 0
    int idx1 = hash(s1, item); // Current location : 0
    Array.set(a0, idx0, 1);    // Current location : 1
    Array.set(a1, idx1, 1);    // Current location : 2
}
```

Figure 4.5: A demonstration of the basic typechecking strategy employed by Lucid. All events start at location 0, and in this case the function `add` does as well; we describe how function starting locations are determined in the next section.

```
1 global Array.t<1> a0 = Array.create(m); // Stored at location 0
2 global Array.t<1> a1 = Array.create(m); // Stored at location 1
3
4 fun void add(int item) {       // Starting location: 0
5     int idx0 = hash(s0, item); // Current location : 0
6     int idx1 = hash(s1, item); // Current location : 0
7     Array.set(a1, idx0, 1);    // Current location : 2
8     Array.set(a0, idx1, 1);    // Error! 0 < 2!
9 }
```

Figure 4.6: A demonstration of how Pipeline Types detect ordering errors.

advance the current location; this is because our abstract pipeline only models stages that contain global values.

Globals can only be accessed if the computation is at or before their stage of the pipeline; if a program tries to access a global after moving past it in the pipeline, typechecking fails. The program in Figure 4.5 typechecks, but supposed the programmer accidentally permuted the array accesses, producing the function shown in Figure 4.6. In this case, Lucid would generate an ordering violation at line 8, since that line accesses `a0`, at location 0, when that location has already been bypassed in the pipeline (at line 7). Given the offending line, the programmer can then simply look backwards from there, notice that they had already accessed `a1` on line 7, and determine a solution. In this case, simply swapping the offending lines would suffice.

Aside: an alternate design choice Lucid demands that all program components access stateful data in the order it is declared. If all components consistently used state in some other order, our system would throw an error even though the program could be compiled. An alternate design could allow programmers to use data in any order, provided they do so consistently across their whole program, or provided the system can permute accesses without changing program semantics to arrive at a consistent order (as was the case in the prior paragraph's example).

This other design is easily achievable and, from a technical perspective, varies little from the one above (we would simply find a satisfying assignment to ordering

Figure 4.7: An ordering error detected by the Lucid compiler, pointing to the exact line of the program that conflicts with the global variable declaration order.

constraints rather than check that such constraints are consistent with an *a priori* ordering). However, we chose to require that programmers follow declaration order for two reasons: (1) declaration order provides useful, built-in documentation and (2) it is easier to provide targeted error messages when things go wrong. Although programmers cannot entirely avoid thinking about state ordering, Lucid boils the requirements down to a simple, easy-to-state guideline: "Use globals in the order they are declared". When programmers violate this guideline, Lucid can issue a simple message of the form "Line X conflicts with the global order," (shown in Figure 4.7) which allows programmers to navigate right to the source of their problem and fix it quickly.

It is also worth noting that the Lucid compiler is free to ignore the declaration order of globals, and store globals in any order it wants, so long as that order is consistent. The purpose of the type system is not to determine the actual order used during compilation, but to ensure that *some* valid order exists.

4.1.3 Polymorphism and Constraints

As in Chapter 3, the Bloom filter code in Figure 4.4 is not reusable: the `add` and `query` functions operate over particular arrays (`a0` and `a1`), whose locations in the pipeline are fixed. In the previous chapter, we saw that we could fix this by allowing the functions to take the arrays as parameters, resulting in code like the following:

```
fun void add(Array.t<1> a0, Array.t<1> a1, int s0, int s1, int item)
{
    int idx0 = hash(s0, item);
    int idx1 = hash(s1, item);
    Array.set(a0, idx0, 1);
    Array.set(a1, idx1, 1);
}
```

However, one cannot guarantee that the code above is safe. Indeed, the function is only safe when the location of `a0` precedes the location of `a1`.

To facilitate proofs of safety, we extend our function definitions to admit **location polymorphism** and **ordering constraints**. Below, we rewrite our function with appropriate constraints, using the special keyword `start` to denote the location at which the function begins execution. Within the constraint clause below, we write `a0 < a1` to mean that $\ell_{a0} < \ell_{a1}$, where ℓ_{a0} and ℓ_{a1} are the locations associated with `a0` and `a1`. The special `end` constraint indicates that the function ends immediately after the designated location.

```
fun void add(Array.t<1> a0, Array.t<1> a1, int s0, int s1, int item)
             [start <= a0 < a1; end a1]
{
    int idx0 = hash(s0, item);
    int idx1 = hash(s1, item);
    Array.set(a0, idx0, 1);
    Array.set(a1, idx1, 1);
}
```

With this change, the typechecker must now ensure that the constraints are met every time the function is called. Doing so requires reasoning about symbolic integer locations and inequality constraints; to do so, we employ the Z3 SMT solver. We describe our SMT encoding in detail in §4.4.3.

Function Types

The Pipeline Type system tracks the location information of a function in that function's type. Function types contain 5 components: input and output types (like regular function types), as well as input and output *locations* (indicating where the function starts and ends), and finally any constraints on the function's arguments.

Function types are written with the following syntax:

$$C \Rightarrow (\tau, \ell) \rightarrow (\tau, \ell)$$

where C is a list of constraints, and τ and ℓ represent (possibly polymorphic) types and locations, respectively. For example, the type of the `add` function in Figures 4.4 and 4.5 would be

$$[\,] \Rightarrow (\texttt{int}, 0) \rightarrow (\texttt{void}, 2)$$

indicating that it has no constraints, should be called at stage 0 in the pipeline, and will finish at stage 2 (one stage after `a1`).

Contrast the above type with the type of our revised `add` function, which is

$$[\alpha < \beta < \gamma] \Rightarrow ((\texttt{Array.t<1>@}\beta * \texttt{Array.t<1>@}\gamma * ... * \texttt{int}), \alpha) \rightarrow (\texttt{void}, \gamma + 1)$$

We use Greek letters to indicate polymorphic variables. The above type states that the function begins executing at α, and takes two arrays at locations β and γ. It finishes executing at location $\gamma + 1$, as indicated by the `end` constraint. However, the function may only be called if the constraints at the beginning hold true, i.e. if $\alpha < \beta < \gamma$.

Like normal polymorphic functions, the polymorphic variables are instantiated separately at each call site. In this case, α will be set to the current location in the pipeline when the function is called, and β and γ will be set to the locations of the

function's array arguments. In order to satisfy the function's constraints, this means that it must be called before either of its array arguments have been used, exactly as one would expect.

4.1.4 Records and Modules

With the addition of location polymorphism, users have the ability to reuse their code across multiple globals – a prerequisite for modularization. The next step is to encapsulate the Bloom filter code by combining the various components into a single record, and wrapping everything up within a module. We have already seen how to do so, and the code is repeated in Figure 4.8.

While extending most languages with compound and abstract types is relatively straightforward, in our case, these extensions have unusual consequences for the structure of the effect system. In particular, consider how we must modify the interface for the Bloom Filter (originally depicted in Figure 3.5). We must extend the annotations on its functions to describe how they interact with the global order; in particular, we must include the function's constraints, as shown in Figure 4.9.

The interface specifies that the `add` and `query` functions should be called before `filter` in the pipeline, and end right after it. In other words, if the filter is at location α, the functions should finish at location $\alpha + 1$.

However, consider how we would typecheck the `add` function, reproduced in Figure 4.10. We must first assign locations to both arrays in the Bloom filter type; the naïve way of doing so is to simply put `a0` at some location α and `a1` at the next location $\alpha + 1$. The function then begins at (or before) location α; when it accesses `a0`, it moves to the next location, and similarly for `a1`. Overall, the function terminates at location $\alpha + 2$.

This type is not the one in the interface! The difference is that this type ends at $\alpha + 2$, rather than $\alpha + 1$. The fundamental problem is that the function type

```
module BloomFilter {
    // An abstract record type, with definition hidden from module clients
    type t = {
        array<1> a0;
        array<1> a1;
        int s0;
        int s1;
    }

    // A compile-time function for creating global values.
    constructor createFilter(int m, int seed0, int seed1) = {
        a0 = Array.create(m);
        a1 = Array.create(m);
        s0 = seed1;
        s1 = seed2;
    }

    // Add item to filter
    fun void add(t filter, int item) {
        int idx0 = hash(filter.s0, item);
        int idx1 = hash(filter.s1, item);
        Array.set(filter.a0, idx0, 1);
        Array.set(filter.a1, idx1, 1);
    }

    // Return true if item in filter
    fun bool query(t<k> filter, int item) {
        int idx0 = hash(filter.s0, item);
        int idx1 = hash(filter.s1, item);
        int<1> b0 = Array.get(filter.a0, idx0);
        int<1> b1 = Array.get(filter.a1, idx1);
        return (b0 == 1 and b1 == 1);
    }
}

// Using the constructor
global filter f1 = BloomFilter.createFilter(...);
global filter f2 = BloomFilter.createFilter(...);
```

Figure 4.8: An abstract, compound type for Bloom filters.

```
interface BloomFilterInterface {
    global type t;
    constructor t create(int m, int seed0, int seed1);

    fun void add  (t filter, int item) [start < filter; end filter];
    fun bool query(t filter, int item) [start < filter; end filter];
}
```

Figure 4.9: An interface for the Bloom filter module that describes how its functions interact with the ordered type system.

```
type t = {
    array<1> a0; // Stored at some location α
    array<1> a1; // Stored at some location α + 1
    int s0;
    int s1;
}

fun void add(t filter, int item) {     // Starting location: α
    int idx0 = hash(filter.s0, item);  // Current location : α
    int idx1 = hash(filter.s1, item);  // Current location : α
    Array.set(filter.a0, idx0, 1);     // Current location : α + 1
    Array.set(filter.a1, idx1, 1);     // Current location : α + 2
}
```

Figure 4.10: A naïve way of typechecking the modularized Bloom filter. Unfortunately, this strategy doesn't match the interface in Figure 4.9.

"leaks" information about the implementation of the module; in particular, the fact that it contains 2 arrays. A module implemented using 3 arrays (`a0`, `a1`, and `a2`) would instead end at $\alpha + 3$. We could in theory create different interfaces for each implementation, but doing so forfeits a major benefit of abstraction: the ability to change implementations without modifying the interface.

Hierarchical Locations

Our solution is to allow locations to be nested, in much the same way that record types allow values to be nested. When we declare a record-type global like a Bloom filter, we create a single "virtual" location ℓ, representing the location of the record as a whole. Each global-typed[3] field of the record is assigned a subordinate location "within" ℓ, written as $\ell.0$ for the first field, $\ell.1$ for the second, $\ell.2$ for the third, and so on. The next global declared will be stored at the next location after ℓ, written $\ell + 1$.

To accommodate these nested locations, we abandon our linear abstract pipeline and replace it with a hierarchical abstract heap. Figure 4.11 depicts the structure of this heap for the program in Figure 4.8, which declares two Bloom filters, `f1` and `f2`.

[3]Non-global fields such as integers are immutable, and need not be stored in persistent memory.

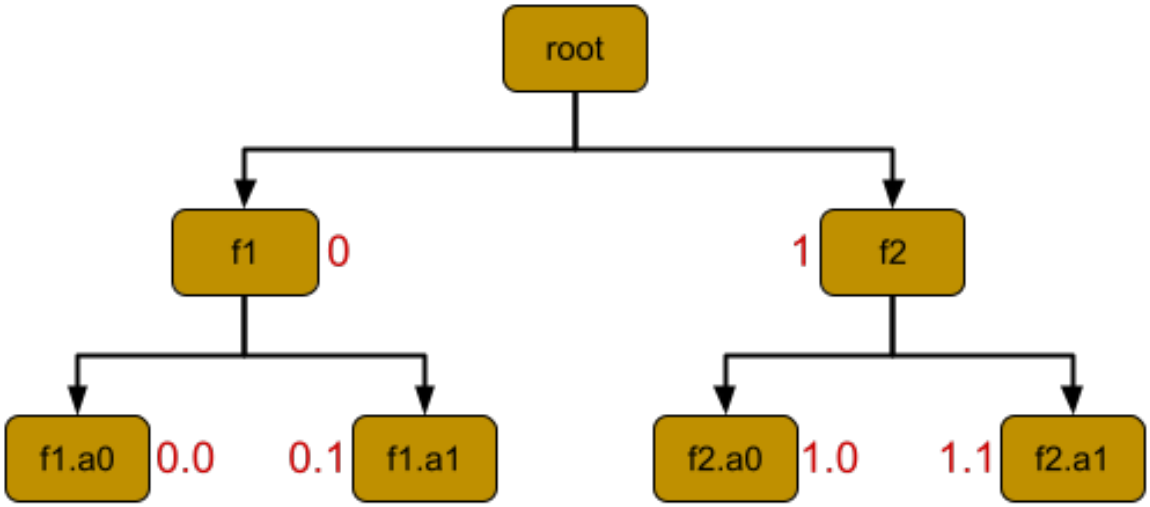

Figure 4.11: An abstract representation of the memory in Figure 4.8. Each node is annotated with its location, which is given by the preorder traversal of the tree.

The heap consists of a single virtual root node, with each global variable represented as a child of the root, from left-to-right in declaration order. Record-type globals induce additional children for each field, also left-to-right in the order those fields were declared in the record type's definition.

We have annotated each node in Figure 4.11 (except the virtual root node) with its location in the abstract heap. These locations correspond precisely to the path to that node from the root. For example `f1` has location 0 (as the 0th child of the root), while `f1.a0` and `f1.a1` have locations 0.0 and 0.1, as the 0th and 1st children of the 0th child of the root, respectively. The global declared after `f1` (`f2`), is assigned location 1, while its children are 1.0 and 1.1, respectively. Generally, each location $n_0.n_1.n_2$, when read from left-to-right, yields the path from the root to that location in the heap.

Successors We define the **successor** of a location ℓ, written $S(\ell)$ or $\ell + 1$, to be the next node on the *same level* as ℓ. Thus the successor of `f1`, at location 0 is `f2`, at location $0 + 1 = 1$. Similarly, the successor of `f1.a0`, at location 0.0, is `f1.a1`, at location 0.1. In general, the successor of a location $a.b...z$ is $a.b...(z + 1)$.

Ordering hierarchical locations To prevent ordering errors, the type system must reason about the order that these locations will ultimately be laid out in a physical pipeline. When comparing locations, we use the preorder traversal of the nodes of the heap. Thus, for instance, `f0` (location 0) appears earlier in the order than `f0.a1` (location 0.1), which is itself earlier than `f1` (location 1). Conveniently, the preorder makes it easy to compare two locations; they are simply compared lexicographically as lists of integers. For example, here is the ordering of several locations:

$$0 < 1 < 1.0 < 1.4.7 < 1.5 < 1.5.3 < 2$$

The preorder provides us with two important properties. First, nodes that are declared first in the program appear earlier in the ordering, exactly as required. Second, because the successor function $S(\ell)$ does not care about the structure of the node at location ℓ, the sucessor of ℓ is always greater than every descendant of ℓ. This means that the successor function can be used to "skip past" compound locations without the knowledge that they are, in fact, compound.

Typechecking Take Two

The preorder properties discussed above provide us with the key to successfully typechecking our modular Bloom filter. The strategy is demonstrated in Figure 4.12. If we have a filter at location ℓ, we assign `a0` and `a1` the locations $\ell.0$ and $\ell.1$, respectively. The `add` function will then move from location ℓ to its sublocation $\ell.2$.

Since we are guaranteed that $S(\ell)$ is greater than any sublocation of ℓ, we can avoid revealing the sublocations to the client by "rounding up" at the end of the function. Thus we assign the function a type that says that it ends at $S(\ell) = \ell + 1$, rather than $\ell.2$, which precisely matches the interface in Figure 4.9. From the perspective of a user outside the module, the `add` function now simply consumes the `filter` argument, moving from location ℓ to $\ell+1$—all information about the implementation

```
type t = {       // Stored at some location ℓ
    array<1> a0; // Stored at some location ℓ.0
    array<1> a1; // Stored at some location ℓ.1
    int s0;
    int s1;
}

fun void add(t filter, int item) {     // Starting location: ℓ
    int idx0 = hash(filter.s0, item);  // Current location : ℓ
    int idx1 = hash(filter.s1, item);  // Current location : ℓ
    Array.set(filter.a0, idx0, 1);     // Current location : ℓ.1
    Array.set(filter.a1, idx1, 1);     // Current location : ℓ.2
    // At the end: "round up" ℓ.2 to ℓ + 1.
}
```

Figure 4.12: Using hierarchical locations to typecheck the `add` function in our Bloom filter module.

of the `filter` type is properly hidden.

Linearizing the heap

One might be concerned that our heap-based view of memory strays too far from the actual, linear pipeline we will be compiling to. However, the existence of a total order on our locations – the preorder – allows us to linearize the heap whenever necessary.

Indeed, note that the leaf nodes of the tree in Figure 4.11 are precisely the array-type variables of the program – that is, the mutable globals that must be stored in the pipeline. This means that it is trivial to assign these arrays to stages in the (abstract) pipeline: simply read the leaf nodes of the tree from left-to-right, and assign them to stages in that order.

4.1.5 Vectors

The final step to fully modularize our Bloom filter is to parameterize it to allow instantiation with different numbers of arrays, using vectors and loops. The code for this is reproduced in Figure 4.13.

Recall that since data-plane programs must ultimately run on the linear switch

```
interface BloomFilterInterface {
    global type t<'k>;
    constr t<'k> create(int m, int[k] seeds);

    fun void add  (t<'k> filter, int item) [start < filter; end filter];
    fun bool query(t<'k> filter, int item) [start < filter; end filter];
}

module BloomFilter : BloomFilterInterface = {
    // An abstract record type, with definition hidden from module clients
    type t<'k> = {
        array<1>[k] arrs;
        int[k] seeds;
    }

    // A compile-time function for creating global values.
    constructor createFilter(int m, int<'k> seeds) = {
        arrs = [Array.create(m) for m < k];
        seeds = seeds;
    }

    // Add item to filter
    fun void add(t<'k> filter, int item) {
        for (i < k) {
            int idx = hash(filter.seeds[i], item);
            Array.set(filter.arrs[i], idx, 1);
        }
    }

    // Return true if item in filter
    fun bool query(t<'k> filter, int item) {
        bpol acc = true;
        for (i < k) {
            int idx = hash(filter.seeds[i], item);
            int<1> b = Array.get(filter.arrs[i], idx);
            acc = acc and (b == 1);
        }
        return acc;
    }
}

// Using the constructor
global filter<2> f1 = BloomFilter.createFilter(1024, [0; 1]);
global filter<3> f2 = BloomFilter.createFilter(1024, [2; 3; 4]);
```

Figure 4.13: A Bloom filter module using vectors and loops to allow instantiations with different values of k

hardware, we allow only bounded loops of the form `for (i < k) { ... }` that can be unrolled during compilation. In order to avoid out-of-bounds errors, we include the length of a vector in its type, and allow indexing operations only if the index can be proved to be in bounds. Constraints generated from an index declaration `i < k` suffice for such proofs in our application domain.

Fortunately, adapting the hierarchical locations of the previous section to accommodate vectors is simple. We can view vectors as nodes in the heap with a variable number of identical children, and when we specify a child we may do so either with a concrete integer as before, or with a loop variable (for example, $0.1.i$ where i is a loop variable). When comparing locations ℓ_1 and ℓ_2 that involve variables, we say that $\ell_1 < \ell_2$ only if that relationship holds for every instantiation of the variables in ℓ_1 and ℓ_2. So, for example, $0.\mathrm{i} < 1$, but 0.i and 0.1 are incomparable.

Loop constraints Since all our loops are bounded, and all vector accesses include bounds checking, termination is guaranteed and indexing errors do not occur. However, we do need to ensure that loop bodies will not result in ordering errors when run multiple times.

To check a loop of the form `for (i < k) { e }` starting at location ℓ_{init}, we must ask:

1. Can we safely execute the loop body with $i = 0$ and starting at ℓ_{init}?
2. For all $j > 0$, can we safely execute the loop body with $i = j$, starting at the ending location of the prior iteration?

To see how we might fail property (1), assume we have two globals of type `Array.t<1>[k]` named `arr1` and `arr2`, at locations 1 and 2, respectively. Assume the function `access` consumes its argument. Now consider the following loop:

```
1 access(arr2[0]);
2 for i < k { access(arr1[i]); }
```

At the start of the loop, ℓ_{init} will be `2.1` (one step past 2.0), and on the first iteration we will access `arr1[0]`, which has location 1.0. Since $1.0 < 2.1$, we run into an ordering error immediately. We can always detect violations of property (1) easily simply by typechecking the loop body with $i = 0$.

Detecting violations of property (2) is trickier. If the loop bound k is an unknown size (e.g. if the loop is inside a size-polymorphic function like `add` or `query`), then naïvely we would need to typecheck the loop body for arbitrarily many iterations, which would require a universally quantified SMT constraint. Unfortunately, it is unclear if the type system is decidable in the presence of universal quantifiers[4].

Fortunately, there is a better way, which becomes apparent after looking at several "bad" loops. Consider the following programs,in which `arr1` and `arr2` are at locations 1 and 2, respectively (their types vary as necessary):

```
1 for i < k {
2   access(arr1[0]);
3 }
4
```

(a)

```
1 for i < k {
2   access(arr1[i]);
3   access(arr2[i]);
4 }
```

(b)

```
1 for i < k {
2   for j < k' {
3     access(arr1[j][i]);
4 } }
```

(c)

Loop (a) will begin at location 0, then access location 1.0 on the first loop. On the second loop, it will try to access location 1.0 again, causing an error. Similarly, Loop (b) will first access locations 1.0 and 2.0, but the second iteration it will try to "go back" to access location 1.1, which is less than 2.0 – another error. Finally, Loop (c) will execute the outer loop once, ending at location 1.k'.1. On the second iteration it will try to access location 1.0.1, which is less than 1.k'.1 (if k' > 0).

The common thread in all these examples is that despite the loops having several different forms, each of the errors occurred very quickly (within a few iterations of the outermost loop). This is not a coincidence; we will prove (in §4.3) that, given certain minor restrictions, *every* "bad" loop will fail in at most three iterations. In other

[4]Typechecking event handlers, which are all mutually recursive, requires us to prove implications of constraints, so the quantifiers are not necessarily at top-level in the SMT formula.

words, if the loop doesn't violate ordering constraints in the first three iterations, it will not do so in any future iteration.

This insight allows us to reduce property (2) from a universal statement to a finite one. Rather than having to reason about every iteration of the loop simultaneously, it suffices to only check the first three. This is a significant victory, and our type system leverages it to turn a potentially undecidable problem into an obviously decidable one.

4.1.6 Location Inference

We have now extended the basic system of Pipeline Types to handle a fully general Bloom filter module, which is configurable in both the number and size of its arrays. However, this did not come entirely without cost – it is only through location inference that we have avoided leaving cumbersome location annotations throughout the program. Inference is crucial for real programs, since it allows the programmer to think at a high level – rather than reasoning about the low-level details of the effect system, they can maintain a simple, high-level invariant.

To support inference, the location grammar we use is carefully designed to have a minimal set of simple constructors: zero (0) and successor ($S(\ell)$) constructors to represent integers, and constant/variable projection operators for record and vector entries ($\ell.0$ and $\ell.i$). This choice means that standard unification algorithms [53] can be directly applied to infer both types and locations. Moreover, we can infer constraints for each expression and function, and for the program as a whole, by collecting them as we walk through the program.

In this way, we have almost entirely eliminated locations from the surface syntax of Lucid. The only places they appear are in the constraints of functions and events, and the user need only write these explicitly in module interfaces (where we do not have function bodies available to run inference), and on events with global-typed

arguments[5]. Through location inference, Lucid programmers are provided with the easy, high level abstraction of "use global variables in the order they are declared", and are not forced to learn a new and technical system before they can continue writing code.

4.2 Formal Type System

In this section, we formalize the system of Pipeline Types that was outlined intuitively in the previous section. To do so, we formally define the syntax of Pipe, an idealized subset of Lucid designed to illustrate and prove correct the central elements of Pipeline Types. We then define an operational semantics for Pipe, as well as a typing judgement, and prove the judgement sound with respect to the operational semantics.

The syntax of Pipe is outlined in Figure 4.14. Like Lucid, it contains a collection of compile-time integers called *sizes*, which are used for describing vector lengths and indices, and may therefore appear in locations[6]. They include constants n (a natural number) as well as two different sorts of identifiers, b and κ. We refer to b as a *bounded size* — our type system ensures that such identifiers will always appear with a constraint $b < k$. Such constraints make vector bounds checking straightforward. We refer to identifiers κ as *unbounded sizes*.

Pipe's type system also includes *locations*, which describe where in the (abstract) pipeline a piece of persistent memory is stored. The metavariable z ranges over concrete locations whereas ℓ ranges over symbolic locations. They use the same syntax as the previous section: the first location in a pipeline is 0, and the location $S(\ell)$ is the successor of the location ℓ. Similarly, ℓ is a location then $\ell.0$ is the first

[5]Inference is complicated for events, since all handlers are mutually recursive, so we require users to supply constraints instead of trying to infer them. However, constraints only refer to global-typed arguments; if an event does not have any (the common case), the user need not write any constraints

[6]*Un*like Lucid, Pipe does not contain sized integers (or any integers at all, for that matter).

indices ι ::= n | b

sizes k ::= ι | κ

concrete locations z ::= 0 | S(z) | z.0

locations ℓ ::= 0 | α | S(ℓ) | ℓ.0 | ℓ.b

constraints C ::= true | $\ell \leq \ell$ | $C \wedge C$

base types T ::= Bool | Unit

raw types t ::= T | addr(T) | (t, t) | vector(t,k)
| $\forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau, \ell) \to (\tau, \ell)$

types τ ::= $t\langle\ell\rangle$

values v ::= () | true | false | fun $[\overline{\kappa}, \overline{\alpha}]$ $(x : \tau, \ell) \to e$
| addr(z)
| (v, v)
| vector($v, \ldots, v$)

expressions e ::= v
| x
| (e, e)
| fst e
| snd e
| vector($e, \ldots, e$)
| e[ι]
| [e for $b < k$]
| !e
| e := e
| let $x = e$ in e
| if e then e else e
| for $b < k$ do e
| e[$\overline{k}$, $\overline{\ell}$] e

Figure 4.14: Formal syntax for our simplified language.

location within ℓ and $S(\ell.0)$ is the next location within ℓ. Symbolic locations can be location variables α or hierarchical locations such as $\ell.b$ where b is an index into ℓ.

Constraints C are conjunctions of inequalities $\ell_1 \leq \ell_2$, which describe the order that locations must appear in memory. There will be more on constraints, locations and operations over them in the following subsection.

Pipe contains `Bool` and `Unit` base types as well as *raw types*, which include function types, vectors with elements of type t and length k (`vector`(t, k)), and pairs (t_1, t_2). Rather than using arrays as the basic mutable type, Pipe includes mutable references (`addr(T)`) for simplicity. There are no references to references (the hardware only admits "flat" data structures); this is why we distinguish "raw types" and "base types." Vectors will be unrolled and their associated contents allocated to stages at compile time; their length k is a compile-time computed value.

Types proper (τ) are pairs of a raw type and the location ℓ where its value is stored, written $t\langle\ell\rangle$. For uniformity in the system, base types like `Bool` and `Unit` are associated with a location even though it is not necessary to do so (the location of a base type winds up playing no role in the system) — only persistent mutable data need be allocated to stage memory.

In general, functions have a type of the form $\forall\overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau_1, \ell_1) \rightarrow (\tau_2, \ell_2)$. These functions are non-recursive, call-by-value functions and will be fully inlined at compile time (the hardware does not have mechanisms for implementing a general purpose function call). They are polymorphic in both sizes (κ), and in locations (α). For simplicity, we do not include type-polymorphic functions in Pipe, although they are present in Lucid; adding them presents no theoretical challenge.

Function preconditions C are a collection of inequality constraints that must be satisfied prior to calling the function. Functions take an argument with type τ_1 and start at location ℓ_1 in the pipeline, returning a result with type τ_2 and completing at location ℓ_2 in the pipeline.

There are values (v) for each type. Notice that function values do not specify required function constraints C — they will be inferred during typechecking.

Expressions contain many standard forms. We often use $e1; e2$ as an abbreviation for `let` $x = e_1$ `in` e_2 when x does not appear free in e_2. Components of a pair are projected using the `fst` and `snd` operators, while vector projection is written $e[\iota]$. The expression $!e$ reads from the address e (analogous to the `Array.get` function in Lucid) and $e_1 := e_2$ writes the value of e_2 to the address e_1 (analogous to `Array.set`).

A vector comprehension [e `for` $b < k$] generates a vector of length k whose i^{th} component is e with b replaced by i. The construction `for` $b < k$ `do` e iterates k times over the body, replacing b with i in the i^{th} iteration. Finally $e_1[\overline{k}, \overline{\ell}]e_2$ calls function e_1 with size vector $\overline{k}$, location vector $\overline{\ell}$ and value e_2 as arguments.

We define capture-avoiding substitution in the usual way, and, for instance, use the notation $e[\ell/\alpha]$ for the expression e with all free occurrences of α replaced with ℓ. We substitute vectors of terms ($\overline{\ell}$) for vectors of variables ($\overline{\alpha}$) using the notation $e[\overline{\ell}/\overline{\alpha}]$. Analogous notation is used to denote other sorts of substitutions. We also treat expressions as equivalent if they differ only in the names of bound variables, which we refer to as "alpha-renaming".

4.2.1 Locations

Location Representations Locations (ℓ) denote (hierarchical) pipeline stages. We have defined the syntax of location expressions (see Figure 4.14) via an algebra that involves a successor function $S(\ell)$, which denotes the location after ℓ. However, an expression like $S(S(S(0.0).k))$ is challenging to understand, and sometimes inconvenient technically (though other times it is quite convenient, especially for unification-based type inference, which is why we chose it). There is an isomorphic notation as a non-empty list of symbolic natural numbers. Such lists have the follow-

ing form:

$$\langle L \text{ } (\textit{list location}) \rangle ::= \iota + n \mid \alpha + n \mid L.(\iota + n)$$

The following function f converts the standard representation of locations ℓ into a list-based representation L.

$$f(0) = 0 \qquad f(\alpha) = \alpha \qquad f(\ell.\iota) = f(\ell).\iota$$

$$f(S(\ell)) = \begin{cases} L.(\iota + n + 1) & \text{if } f(\ell) = L.(\iota + n) \\ f(\ell) + 1 & \text{otherwise} \end{cases}$$

For example, if we apply f to $S(S(S(0.0).i))$ we get the list $0.1.(i + 2)$. We use standard list syntax to refer to elements; in our previous example, the head would be 0 and the tail would be $1.(i + 2)$. The function f is bijective, so either location syntax contains the same information. In a slight abuse of notation, from this point forward, we will implicitly convert locations back and forth between representations, using whichever is most convenient at the time. We will use the metavariable ℓ to range over effects regardless of the representation.

Location Ordering When location ℓ_1 occurs earlier in a pipeline than ℓ_2, we write $\ell_1 < \ell_2$. In general, $\ell_1 < \ell_2$ is defined (using the list-based representation of locations) as follows: $\ell_1 < \ell_2$ iff:

1. ℓ_1 is an empty list and ℓ_2 is a non-empty list[7], or
2. `hd` $\ell_1 <$ `hd` ℓ_2, or
3. `hd` $\ell_1 =$ `hd` ℓ_2 and `tl` $\ell_1 <$ `tl` ℓ_2

If either list contains variables (αs, κs, or bs), we say $\ell_1 < \ell_2$ if and only if that relationship holds for all possible instantiations of the variables. That is, we have $0.0 < 0.(i + 1)$, but 0.1 and $0.i$ are be incomparable.

[7]Although the output of f will never be empty, we may generate an empty list while checking inequality by use of the `tl` operator.

Location Rounding When processing symbolic locations, we sometimes wish to jump forward to a location guaranteed to come after the symbolic location. For example, given the location $0.0.b$, we may want to jump to 0.1, which is "ahead" of (i.e. greater than) $0.0.b$, for all b. We call this operation *rounding*, and write it $\texttt{round}(\ell, b)$.

We define `round` in terms of another function, `drop`, which simply drops all entries after the first instance of b it encounters. Below, and elsewhere, we use the notation $b \notin \ell$ to indicate that ℓ does not contain any instances of b.

$$\texttt{round}(\ell, b) = \begin{cases} \ell & b \notin \ell \\ S(\texttt{drop}(\ell, b)) & \text{otherwise} \end{cases}$$

where $\texttt{drop}(\ell, b) = \ell$ if $b \notin \ell$, and otherwise

- $\texttt{drop}(S(\ell), b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.0, b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.b, b) = \texttt{drop}(\ell, b)$
- $\texttt{drop}(\ell.b', b) = \texttt{drop}(\ell, b)$

Location Well-formedness The predicate $\texttt{nri}(\ell, b)$ is true when ℓ contains no more than one instance of b (`nri` stands for "no repeated index"). The predicate $\texttt{nri}(\ell)$ is true when ℓ contains no more than one instance of any single b. Finally, $\texttt{nri}(C)$ is true when all locations ℓ appearing in C satisfy $\texttt{nri}(\ell)$.

Constraints We write $C \Rightarrow C'$ to mean that C implies C', and we write $\vDash C$ when C is *valid* — i.e., for all well-typed substitutions of values for variables, C is satisfied.

4.2.2 Pipeline Semantics

Our operational model captures execution of expressions on an abstract pipelined processor. In this model, computations must be organized so that they access memory locations in order, possibly skipping over some of the locations they do not need to access. Immediately after a computation accesses a location, the state of the machine is advanced — hence, each location is accessed at most once.

In a real PISA architecture, such as the Intel Tofino [1], a single atomic action may involve several operations, such as a read, a conditional test and a write to the same state that was read from, but successive atomic actions may not touch the same state. Augmenting our machine model with additional primitives to model such compound operations is straightforward; indeed, they are present in Lucid as memops. However, the abstraction we present in Pipe, with its simplified atomic actions, captures the essence of such computations.

Formally, the states of our abstract machine are triples (M, z, e), where M is a **pipelined memory**, z is our current location in the memory, and e is the expression to execute. A pipelined memory is a partial mapping from concrete locations to values.

Figure 4.15 presents selected rules from the small-step operational semantics of these machines as a relation with the form $(M, z, e) \rightarrow (M', z', e')$. The complete semantics appears in Appendix A.1.

The most interesting rules are DEREF-2 and UPDATE-3. Given that the current location is z and the computation requests a read from address z_e, DEREF-2 states that the machine skips forward to z_e (which must be later in the ordering than z), reads the value in memory at that location, and then advances the current location to $S(z_e)$. UPDATE-3 is similar— the machine skips forward from z to z_e, writes to z_e and then moves forward to the successor location $S(z_e)$.

There are a number of ways such stateful computations can "go wrong." The

$$\textsc{Deref-1}\quad \frac{M, z, e \to M', z', e'}{M, z, !e \to M', z', !e'} \qquad \textsc{Deref-2}\quad \frac{z \leq z_e}{M, z, !\texttt{addr}(z_e) \to M, S(z_e), M[z_e]}$$

$$\textsc{Update-1}\quad \frac{M, z, e_1 \to M', z', e_1'}{M, z, e_1 := e_2 \to M', z', e_1' := e_2} \qquad \textsc{Update-2}\quad \frac{M, z, e \to M', z', e'}{M, z, v := e \to M', z', v := e'}$$

$$\textsc{Update-3}\quad \frac{z \leq z_e}{M, z, \texttt{addr}(z_e) := v \to M[z_e := v], S(z_e), ()}$$

$$\textsc{Vector}\quad \frac{M, z, e_0 \to M', z', e_0'}{M, z, \texttt{vector}(v_0, \ldots, v_n, e_0, \ldots, e_m) \to M', z', \texttt{vector}(v_0, \ldots, v_n, e_0', \ldots, e_m)}$$

$$\textsc{Index-1}\quad \frac{M, z, e \to M', z', e'}{M, z, e[n] \to M', z', e'[n]} \qquad \textsc{Index-2}\quad \frac{n \leq m}{M, z, \texttt{vector}(v_0, \ldots, v_m)[n] \to M, z, v_n}$$

$$\textsc{Loop}\quad \frac{}{M, z, \texttt{for } b < n \texttt{ do } e \to M, z, e[0/b]; ...; e[n-1/b]; ()}$$

$$\textsc{Comp}\quad \frac{}{M, z, [e \texttt{ for } b < n] \to M, z, \texttt{vector}(e[0/b], \ldots, e[n-1/b])}$$

$$\textsc{App-1}\quad \frac{M, z, e_1 \to M', z', e_1'}{M, z, e_1\ [\overline{k}, \overline{\ell}]\ e_2 \to M', z', e_1'\ [\overline{k}, \overline{\ell}]\ e_2} \qquad \textsc{App-2}\quad \frac{M, z, e_2 \to M', z', e_2'}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ e_2 \to M', z', v_1\ [\overline{k}, \overline{\ell}]\ e_2'}$$

$$\textsc{App-3}\quad \frac{v_1 = \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}]\ (x : \tau, \ell) \to e_{body}}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ v_2 \to M, z, e_{body}[v_2/id][\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

Figure 4.15: Pipeline Semantics

location z_e might not exist. If it does, it might not be later in the ordering than the current location z (*i.e.*, we might have already passed it in the pipeline). Our type system will have to present such scenarios from arising.

Also of note are the operational rules for vectors and loops. In particular, at run time, a loop bounded by n may be unrolled to n copies of its body. A key goal of the type system will be to prove such an unrolling is safe—that execution of n copies of the loop body in sequence will not cause an ordering error.

4.2.3 Type Checking

The central goal of the type system is to ensure that the stages of the pipeline are accessed in order, though there are auxiliary goals as well, such as ensuring that vectors are not indexed out of bounds and that operations are applied to arguments of appropriate type.

Typing Environments

The typing environment, $\Omega = (\mathbb{G}, \Delta, \mathbb{K}, \Gamma)$, consists of:

- $\mathbb{G}$, the global persistent state, a partial map from concrete locations z to base types;
- Δ, a set of location and unbounded size variables (αs and κs) that are currently in scope;
- $\mathbb{K}$, a mapping from bounded sizes b to their upper bound, a size (with $\mathbb{K}$ written as a sequence of inequalities $b_1 < k_1, \ldots, b_n < k_n$); and
- Γ, a mapping from value identifiers to types.

We often refer to part of the environment using dot notation (*e.g.*, $\Omega.\mathbb{G}$). We use the notation $\Omega.(...)$ to denote Ω with one of its fields replaced by the body of the parentheses, e.g. $\Omega.(\Delta \cup \Delta')$ replaces Δ with $\Delta \cup \Delta'$. We use the metavariable Σ to range over environments in which all but the first entry are empty; that is, Σ is an environment with the form $(\mathbb{G}, \emptyset, \emptyset, \emptyset)$.

Well-Formedness

The locations, sizes and types manipulated by the type checker must be well-formed, that is, any free variables must be declared in the type checking environment. We write $\Delta, \mathbb{K} \vdash k$ and $\Delta, \mathbb{K} \vdash \ell$ when the free variables of k and ℓ are contained in Δ and the domain of $\mathbb{K}$. We say $\mathbb{K}$ is well-formed with respect to Δ, written $\Delta \vdash \mathbb{K}$ under the following conditions.

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \vdash \mathbb{K} \qquad b \notin \mathsf{Dom}(\mathbb{K}) \qquad \Delta, \mathbb{K} \vdash k}{\Delta \vdash \mathbb{K}, b < k}$$

We use similar notation (*e.g.*, $\Delta, \mathbb{K} \vdash t$, $\Delta, \mathbb{K} \vdash \tau$, and $\Delta, \mathbb{K} \vdash \Gamma$) to describe well-formedness of other objects. Likewise, we write $\Omega \vdash k$ when $\Omega.\Delta, \Omega.\mathbb{K} \vdash k$ and again similarly for other objects. The formal definition is standard; the complete set of well-formedness rules appears in Appendix A.2.

We impose additional well-formedness conditions on function types. The conditions represent useful properties of the type system, which we wish to ensure are respected by any type annotations in the program. The conditions are not strictly necessary — allowing programs with ill-formed type annotations would not violate soundness — but enforcing the conditions allows us to prove properties of the system modularly.

Definition 4.2.1 (Well-Formed Types)**.** If $t = \texttt{fun}\ \forall \overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \to (\tau_{out}, \ell_{out})$,

in order to show $\Omega \vdash t$ we additionally require that

- (monotonicity) C_f implies the constraint $\ell_{in} \leq \ell_{out}$; that is $C_f \Rightarrow \ell_{in} \leq \ell_{out}$, and
- (well-constrained) For every constraint $x \leq y$ in C_f, $C_f \Rightarrow \ell_{in} \leq x \leq y \leq \ell_{out}$.

We impose an additional well-formedness condition on $\mathbb{G}$ as well. Intuitively, $\mathbb{G}$ represents the locations in memory where values are stored; that is, $\mathbb{G}$ should contain entries for each leaf node in the heap. For example, a $\mathbb{G}$ representing the heap in figure 4.11 would have four entries: 0.0, 0.1, 1.0, and 1.1. Our well-formedness condition requires that no entry in $\mathbb{G}$ is a parent or child of another entry. If $\mathbb{G}$ did contain two entries, one a parent of the other, then intuitively the data in those two entries would "overlap." Such constructions do not conform to our mental model of how heaps should be structured and do not arise in practice, though admitting such artificial structures would not compromise the soundness of the system.

Definition 4.2.2 (Well-Formed Globals)**.** A global map $\mathbb{G}$ is well-formed, written $\vdash \mathbb{G}$, if for any two concrete locations z_1, z_2 where z_1 is a strict prefix of z_2, at most one of $\mathbb{G}[z_1], \mathbb{G}[z_2]$ exists.

Constructing Global Maps

In the rest of this chapter, we assume that global maps $\mathbb{G}$ are simply handed to us. However, when checking real programs, we must construct the maps ourselves. Fortunately, we can do so easily by processing global declarations one-by-one at the beginning of the program. For example, to construct the map for a program that begins with

```
1 global int g1 = ...;
2 global (int, bool) g2 = ...;
3 global int[4] g3 = ...;
```

we would add entries for the locations 0 (for `g1`); 1.0 and 1.1 (for `g2`); and 2.0, 2.1, 2.2, and 2.3 (for `g3`). Note how this map adheres to our well-formedness condition.

Expression Typing

The typing judgement for expressions has the form $\Omega, \ell_{in} \vdash e : \tau, \ell_{out}, C$. Here, τ is the type of expression e, ℓ_{in} denotes our place in the pipeline prior to execution of e, while ℓ_{out} denotes our place in the pipeline after execution of e. C contains any ordering constraints required for e to be safe to execute. Figures 4.16 and 4.17 present the typing rules.

Part 1: Values, Functions, and Conditionals Figure 4.16 presents the rules for values, variables, pairs, functions, `let` expressions and `if` statements. Notice that the beginning and ending locations for values are always the same—they have no effect on the state of the pipeline. For uniformity, base types (`Unit` and `Bool`), are associated with a location ℓ'. However, these locations are artificial—only mutable globals need be assigned a stage for storage—and hence the location assigned may be arbitrary. On the other hand, the global stored at address `addr(`z`)` (see rule ADDR) is given a type that includes its location. Values may appear anywhere and hence never directly give rise to any ordering constraints (the generated constraints C are always simply `true`).

Pairs, `let` expressions and `if` statements all involve execution of multiple expressions, and may see the current pipeline location advance from ℓ_0 to ℓ_1 to ℓ_2, *etc.*, as subexpressions are executed. The resulting location of an `if` statement is the greater of the two locations of its branches (locations will be bypassed if one branch uses a location and another does not).

Functions abstract over polymorphic location and size variables and capture the constraints a caller must satisfy to call them. Rules ABS and APP are relatively

$$\textsc{Unit}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash () : \texttt{Unit}\langle \ell' \rangle, \ell, \texttt{true}} \qquad \textsc{True}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{true} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}}$$

$$\textsc{False}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{false} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}} \qquad \textsc{Addr}\ \frac{\Omega.\mathbb{G}[z] = T}{\Omega, \ell \vdash \texttt{addr}(z) : \texttt{addr}(T)\langle z \rangle, \ell, \texttt{true}}$$

$$\textsc{Var}\ \frac{\Omega.\Gamma[id] = \tau}{\Omega, \ell \vdash id : \tau, \ell, \texttt{true}} \qquad \textsc{Pair}\ \frac{\Omega, \ell_0 \vdash e_1 : t_1\langle \ell.0 \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : t_2\langle \ell.1 \rangle, \ell_2, C_2}{\Omega, \ell_0 \vdash (e_1, e_2) : (t_1, t_2)\langle \ell \rangle, \ell_2, C_1 \wedge C_2}$$

$$\textsc{Fst}\ \frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{fst}\ e : t_1\langle \ell.0 \rangle, \ell_1, C_1} \qquad \textsc{Snd}\ \frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{snd}\ e : t_2\langle \ell.1 \rangle, \ell_1, C_1}$$

$$\textsc{Let}\ \frac{\Omega, \ell_0 \vdash e_1 : \tau_1, \ell_1, C_1 \qquad \Omega.(\Gamma[id := \tau_1]), \ell_1 \vdash e_2 : \tau_2, \ell_2, C_2}{\Omega, \ell_0 \vdash \texttt{let}\ id = e_1\ \texttt{in}\ e_2 : \tau_2, \ell_2, C_1 \wedge C_2}$$

$$\textsc{If-left}\ \frac{\begin{array}{c}\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle \ell \rangle, \ell_1, C_1 \\ \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_2 \leq \ell_3\end{array}}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_3, C_1 \wedge C_2 \wedge C_3}$$

$$\textsc{If-right}\ \frac{\begin{array}{c}\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle \ell \rangle, \ell_1, C_1 \\ \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_3 \leq \ell_2\end{array}}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_2, C_1 \wedge C_2 \wedge C_3}$$

$$\textsc{Abs}\ \frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \qquad \Delta' = \Omega.\Delta \cup \overline{\kappa} \cup \overline{\alpha} \\ \Delta', \mathbb{K} \vdash \tau_{in}, \ell_{in} \qquad (\mathbb{G}, \Delta', \mathbb{K}, \Gamma[id := \tau_{in}]), \ell_{in} \vdash e : \tau_{out}, \ell_{out}, C \\ t_f = \forall \overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau_{in}, \ell_{in}) \to (\tau_{out}, \ell_{out}) \qquad \Omega \vdash \ell' \qquad \Omega \vdash t_f\end{array}}{\Omega, \ell \vdash \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}](id : \tau_{in}, \ell_{in}) \to e : t_f\langle \ell' \rangle, \ell, \texttt{true}}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Omega \vdash \overline{k}, \overline{\ell} \qquad \Omega, \ell_0 \vdash e_1 : t_f\langle \ell' \rangle, \ell_1, C_1 \\ t_f = \forall \overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \to (\tau_{out}, \ell_{out}) \qquad \Omega, \ell_1 \vdash e_2 : \tau_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_2, C_2\end{array}}{\Omega, \ell_0 \vdash e_1\ [\overline{k}, \overline{\ell}]\ e_2 : \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], C_1 \wedge C_2 \wedge C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \wedge \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

Figure 4.16: Expression Typing: Values, Conditionals, Functions

standard (if notationally intensive), although note the last constraint of the App rule, which allows locations to be skipped to match the function's input location.

Part 2: State, Vectors, and Loops Figure 4.17 presents rules for checking state, vectors and loops.

In the Deref rule, the current location has advanced to ℓ_1 just prior to derefence. Hence, one must prove the address accessed (ℓ_2) appears later than ℓ_1 in the pipeline – this is the constraint added in the conclusion of the rule. After execution of the expression, the current location will be the successor of ℓ_2. Because the value returned from the read has a base type, the location ℓ' associated with it is irrelevant and may be chosen arbitrarily. The Update rule follows a similar pattern.

When checking indexing operations, the key is to ensure indices are in bounds. Fortunately, patterns for using vectors in Lucid programs are limited, so simple bounds checking rules suffice. The rule Index-const allows constants to be used to index vectors of known length and checks that the index n is less than the vector length n'. In rule Index-var, variables b may index vectors only when the bound on b (given by $\mathbb{K}$) is equal to the length of the vector. This latter rule allows simple loops to iterate over vectors one location at a time, the common case in our experience. Notice that these rules do not manipulate the ending location, because vectors are not themselves global values.

The most interesting rules are the rules for loops (Loop) and comprehensions (Comp). The Loop rule analyzes the loop body e, as if it starts from some arbitrary location α_{start} and with respect to a loop index variable b. Doing so generates a collection of constraints C that is parametric in α_{start} and b. Three instances of C are then created, C_0, C_1, and C_2, representing the constraints that would be generated on the 0^{th}, 1^{st}, and 2^{nd} iterations of the loop. The premise `nri(`C`,`b`)` requires that all locations ℓ appearing in C contain at most one occurrence of b (for example, the

$$\textsc{Deref}\quad \frac{\Omega, \ell_0 \vdash e : \texttt{addr}(T)\langle \ell_2 \rangle, \ell_1, C \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash !e : T\langle \ell' \rangle, S(\ell_2), C \wedge \ell_1 \leq \ell_2}$$

$$\textsc{Update}\quad \frac{\Omega, \ell_0 \vdash e_1 : \texttt{addr}(T)\langle \ell_3 \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : T\langle \ell \rangle, \ell_2, C_2 \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash e_1 := e_2 : \texttt{Unit}\langle \ell' \rangle, S(\ell_3), C_1 \wedge C_2 \wedge \ell_2 \leq \ell_3}$$

$$\textsc{Vector}\quad \frac{\Omega, \ell_0 \vdash e_1 : t\langle \ell_v.0 \rangle, \ell_1, C_1 \qquad \cdots \qquad \Omega, \ell_{n-1} \vdash e_n : t\langle \ell_v.(n-1) \rangle, \ell_n, C_n}{\Omega, \ell_0 \vdash \texttt{vector}(e_1, \ldots, e_n) : \texttt{vector}(t, n)\langle \ell_v \rangle, \ell_n, C_1 \wedge \cdots \wedge C_n}$$

$$\textsc{Index-const}\quad \frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, n')\langle \ell \rangle, \ell_1, C \qquad n < n'}{\Omega, \ell_0 \vdash e[n] : t\langle \ell.n \rangle, \ell_1, C}$$

$$\textsc{Index-var}\quad \frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, k)\langle \ell \rangle, \ell_1, C \qquad \Omega.\mathbb{K}[b] = k}{\Omega, \ell_0 \vdash e[b] : t\langle \ell.b \rangle, \ell_1, C}$$

$$\textsc{Loop}\quad \frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \\ \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : \tau, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash \texttt{for}\ b < k\ \texttt{do}\ e : \texttt{Unit}\langle \ell \rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

$$\textsc{Comp}\quad \frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \\ \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : t\langle \ell_v.b \rangle, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash [e\ \texttt{for}\ b < k] : \texttt{vector}(t, k)\langle \ell_v \rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

Figure 4.17: Expression Typing: State, Vectors, Loops

location $0.b.1.b$ would be disallowed; see §4.2.4 for a more detailed explanation). So long as this is true, it suffices to only check C_0, C_1 and C_2. If they are consistent, then the loop is safe to execute—there will be no ordering violations regardless of the number of iterations of the loop at run time. We sketch the proof of this property in §4.3; a full proof can be found in Appendix A.3 (the Loop Unrolling Lemma).

To determine the current location after execution of the loop, we take the effect at the end of the loop body, $\ell_{end}[\ell_{init}/\alpha_{start}]$, and we "round up" past b. For instance, if we were just iterating over locations 0.0.0, 0.0.1, 0.0.2, ... *etc.*, which are all captured parametrically as $0.0.b$, then this rounding operation advances us past all such indices to location 0.1 by "rounding up," or chopping off everything after b and moving to the successor location.

The Comp rule governs type checking of vector comprehensions. It too is an iterative construct and hence inherits much of the complexity of the Loop rule.

4.2.4 Limitations

Like most type systems, Pipeline Types are incomplete: there exist programs that execute without error, but which fail to type check. One example of incompleteness arises while checking `if` statements. Expressions like the following one will not type check when the relation between locations of `x` and `y` is unknown.

```
1  if ... then !x else !y
```

We have considered adding a "max" operator to serve as a join for our algebra of locations (`max(`ℓ_1`,`ℓ_2`)` being the larger of the two locations), but doing so appeared to complicate type inference, and did not appear worth the effort; in practice, we have not yet encountered any applications that would benefit from such an extension.

One other source of incompleteness arises in the Loop and Comp rules, where the premise `nri`(C, b) rules out programs that use the same index variable twice, as in the expression `g[i][i]`. The following program fragment demonstrates why this

is necessary:

```
1 for i < 10 {
2   !g[i][i]; // Double indexing -- eventually we'll try to access g[6][6]
3   !g[i][5]; // Single indexing -- eventually we'll try to access g[6][5]
4 }
```

This program would succeed for the first five iterations, but fail on the sixth. That is, it is *not* sufficient to check only the first three iterations of this loop. The $\texttt{nri}(C, b)$ premise serves to weed out these examples. This restriction does rule out some legitimate programs – e.g. the above example with line 3 commented out. However, while there are applications that iterate through elements of a vector, we have not seen any that iterate along a diagonal like this. So again, this limitation does not appear to have any practical impact.

4.3 Properties of Pipe

In this section, we discuss selected properties of Pipe and its type system, and finish with a statement of soundness. Proofs of each property are available in Appendices A.4 (soundness) and A.3 (everything else).

Value Lemma. The following lemma states that values are inert; they do not have an effect on the world or generate constraints. They can appear anywhere in the pipeline.

Lemma 4.3.1 (Value Lemma)**.** If $\Omega, \ell \vdash v : \tau, \ell', C$, then

- (V-1) $\ell = \ell'$ and $C = \texttt{true}$.
- (V-2) For all ℓ, we have $\Omega, \ell \vdash v : \tau, \ell, C$.

Location Weakening. Intuitively, the following lemma states that if we can type-

check an expression from a given location, we can also typecheck it from any earlier location. This is exactly as we would expect, since starting execution from an earlier location in the pipeline gives us access to all the same data as before.

Lemma 4.3.2 (Location Weakening)**.** Assume that $\vdash \Omega$ and $\Omega, \ell_{start}, \vdash e : \tau, \ell_{end}, C$ where $\vDash C$. Then, for all $\ell'_{start} \leq \ell_{start}$, there is some $\ell'_{end} \leq \ell_{end}$ such that $\Omega, \ell'_{start}, \vdash e : \tau, \ell'_{end}, C'$, where $\vDash C'$. Furthermore, either $\ell'_{end} = \ell_{end}$ or $\ell'_{end} = \ell'_{start}$.

Monotonicity. When the constraints generated from an expression hold, computations are guaranteed to move forward in the pipeline. The monotonicity property establishes this fact.

Lemma 4.3.3 (Monotonicity)**.** If $\vdash \Omega$, and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then $C \Rightarrow \ell_{start} \leq \ell_{end}$.

Bounded Constraints. The following lemma is the first step in proving properties of loops. It allows us to connect the starting and ending location of a typing judgement with the constraints generated by that judgement.

Lemma 4.3.4 (Bounded Constraints)**.** If $\vdash \Omega$, and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then for each constraint $x \leq y \in C$ we have $C \Rightarrow \ell_{start} \leq x \leq y \leq \ell_{end}$.

Loop Unrolling. If a loop survives three iterations, it will survive arbitrarily many more; the following lemma is key to proving this fact. Since it is such an important property, we provide a high-level proof sketch here as well as the statement of the lemma.

Lemma 4.3.5 (Loop Unrolling)**.** Assume that $\vdash \Omega$ and $\Omega, \alpha_{start} \vdash e : \tau, \ell_{end}, C$. For all locations ℓ_{init} and bounded sizes i, define $\ell_0 = \ell_{init}$, $C_0 = C[\ell_0/\alpha_{start}][0/i]$ and for $j > 0$ define $\ell_j = \ell_{end}[\ell_{j-1}/\alpha_{start}][(j-1)/i]$ and $C_j = C[\ell_j/\alpha_{start}][j/i]$. Finally, assume $\texttt{nri}(C,\ i)$. Then, if M is a model of $C_0 \wedge C_1 \wedge C_2$, M is also a model of $\forall j \geq 0.C_j$.

We prove this lemma by fixing a model M, then showing that for each constraint $x \leq y \in C$, $x[j/i] \leq y[j/i]$ for all $j > 0$. To do so, we use the fact that the initial location of loop iteration $j+1$ is the same as the final location of iteration j. Together with the Bounded Constraints lemma, this lets us conclude that $x[j/i] \leq y[j/i] \leq x[j+1/i] \leq y[j+1/i]$, so long as we know that the left- and right-most inequalities hold separately. We know they do when $j = 1$, since M satisfies C_1 and C_2, and so we use the fact that $y[1/i]$ is "sandwiched" between $x[1/i]$ and $x[2/i]$ (and similarly for $x[2/i]$) to perform a case analysis on the structure of x and y that shows the inequality will always hold regardless of j.

An astute reader might wonder why we chose to use C_1 and C_2 rather than C_0 and C_1. This stems from the fact that the initial location of the loop iteration may appear in constraints, and may not always have the same form between iterations; if it does not, the sandwiching technique fails. While the initial location of each iteration after the first follows a set pattern, the initial location of the first iteration is determined by the code *before* the loop, and hence may differ from the following iterations. Thus we can relate the initial locations of iterations 1 and 2, but not of iterations 0 and 1. This may be a limitation of our proof technique, as in practice, we know of no loops that succeed for two iterations but fail on the third. However, it is not a costly limitation—our type checker can analyze any of our benchmarks in under two seconds (§4.4.3).

Memory Typing. Execution through the pipeline will proceed without error provided the state associated with the pipeline has the expected structure. The following definition describes the required relation between memories M and global specifications $\mathbb{G}$. When the $\mathbb{G}$ in question is clear from context, we may omit it and simply say "M is well-formed."

Definition 4.3.1. M is well-formed with respect to $\mathbb{G}$, written $M \sim \mathbb{G}$, when it satisfies the following properties.

- $M[z]$ exists if and only if $\mathbb{G}[z]$ exists, and
- If $M[z] = v$ and $\mathbb{G}[z] = T$ then for all Ω, ℓ, ℓ', we have $\Omega, \ell \vdash v : T\langle\ell'\rangle, \ell, \texttt{true}$

Soundness. The prior lemmas constitute the scaffolding upon which we can prove a soundness theorem based on progress and preservation. The proofs of these theorems appear in Appendix A.4.

Theorem 4.3.1 (Progress)**.** Let $\Sigma, z \vdash e : \tau, z', C$ where $\vDash C$. Let $M \sim \Sigma.\mathbb{G}$. Then either e is a value or there are some M', z'', e' such that $M, z, e \rightarrow M', z'', e'$.

Theorem 4.3.2 (Preservation)**.** Let $\Sigma, z_{start} \vdash e : \tau, z_{end}, C$ and $M, z_{start}, e, \rightarrow M', z_{step}, e'$, where $\vDash C$ and $M \sim \Sigma.\mathbb{G}$. Then $M' \sim \Sigma.\mathbb{G}$, and $\Sigma, z_{step} \vdash e' : \tau, z'_{end}, C'$, where $\vDash C'$ and $z'_{end} \leq z_{end}$.

4.4 Implementation

In this section, we briefly describe our implementation of Lucid's type system in OCaml. While Pipe's type system presents the core of Pipeline Types, our practical implementation has a number of differences, beyond the obvious fact that it describes

a larger language and must handle additional constructs.

The primary difference is that Lucid does not contain `addr`s, nor does it have the `!` or `:=` operators. Instead, Lucid has *built-in global types*, which are modeled as built-in libraries. The most fundamental is the Array library, which contains the array type and operations for accessing it. Those operations (e.g. `get` and `set`) are presented as entirely normal functions whose type indicates that they consume their array argument. By calling these functions, programs move forward through the pipeline even though Lucid doesn't provide any primitives for doing so.

Similarly, Pipe does not contain recursive functions; while the same is technically true for Lucid, the ability of events to generate other events allows programmers to implement recursive programs. As such, all event handlers are essentially mutually recursive with each other, complicating type inference. We expand upon this challenge in the next subsection.

Smaller differences include the fact that Lucid allows type polymorphism in addition to size and location polymorphism, and performs type inference so that functions need not be annotated with their constraints (although programmers may still do so, either as a reminder or to impose additional, artificial constraints). Lucid also the takes the opportunity to make optimizations that don't appear in Pipe's formal system. For example, the implementation makes use of OCaml's mutable `ref` cells to speed up type inference. It also checks whether the constraints on a function are actually satisfiable when the function is defined – while it is perfectly legal to write a function with contradictory constraints, such a function could never be called, and thus presumably indicates a mistake on the programmer's part.

4.4.1 Typechecking Handlers

Our formal language omits recursion, and our implementation is similar, since the switch hardware cannot implement unbounded recursion in a single pass through a

pipeline. However, recursive programs can be implemented via the packet recirculation mechanism available on the Tofino chip, which directs packets exiting the chip back to the beginning of the pipeline. Recirculation is made available to programmers via events and event handlers, and hence, event handlers are effectively mutually recursive with one another, drastically complicating constraint inference.

Rather than attempting to infer constraints for handlers, we opted to require user-supplied constraint annotations when events are declared. We check that the constraints hold whenever a new event of the given type is generated, and assume the constraints in the body of the event handler when it receives such an event. For instance, we might declare an event `foo` as follows.

```
1 event foo(array<bool> x, array<bool> y) [x < y];
```

Doing so mandates the system prove `x < y` whenever a `foo` event is generated, and allows the `foo` handler to assume `x < y`. In other words, these events are a form of dependent pair.

These constraints place some annotation burden on the programmer, but the burden is minimal and the explicit annotations serve as useful documentation. In practice, many events do not require constraint annotations at all – they are only required when an event takes global variables as parameters, which is rare. In most cases, we can typecheck the body without any assumptions about the order of the parameters.

4.4.2 Type Inference

Lucid's primary type inference algorithm is a generalization of Hindley-Milner type inference [53] that uses mutable `ref` cells to short-circuit costly generalization steps. The algorithm is based on the description given in by Kiselyov [46], although his "level" system ended up being overkill for Lucid due to its lack of recursive functions.

Internally, the "raw" type of each expression is represented using the `raw_ty`

```
type tyvar =
    | Unbound of id
    | Link of raw_ty

and raw_ty =
    | TVar of tyvar ref
    | QVar of id
    | TInt of size
    | TBool
    | TTuple of raw_ty list
    | ...
```

Figure 4.18: A snippet of the OCaml code defining the structure of types.

```
type size =
  | IVar of size tqvar
  | IConst of int
  | IUser of cid (* User-defined size *)
  (* Normal form: list is non-empty, sorted, and
     no entries are Link, IConst, or ISum *)
  | ISum of size list * int


type effect =
  | FVar of effect tqvar
  | FZero                  (* Start of the pipeline *)
  | FProj of effect        (* Nested locations, e.g. 0.1 *)
  | FIndex of id * effect  (* Nested locations with indices, e.g. 0.j *)
  | FSucc of effect        (* Successor *)
```

Figure 4.19: The grammar used to define sizes and effects in our implementation of pipeline types.

datatype defined in Figure 4.19[8]. A raw type may be a regular type (integer, boolean, tuple, array), or a type variable (`TVar` or `QVar`). Regular type variables (`tyvar`) may be either unbound (their initial state when created), or they may be a pointer to another raw type. Quantified type variables (`QVar`) are used for representing univerally quantified types in function definitions.

Unification When our type inference algorithm determines two expressions must have the same type (e.g. if they are being compared for equality), those types are **unified**. Unification involves comparing the raw types; if those differ, the unification fails. If we ever try to compare a `TVar` against another raw type, we instead modify the `TVar` to point at that other type (if it is `Unbound`), or compare with the type it points to (if it is a `Link`). Since `TVar`s are pointers, modifying it affects every copy of that `TVar` in the program, speeding up type inference.

Sizes and Effects This strategy is also utilized for locations (internally referred to as "effects") and sizes, each of which gets their own variant of `TVar` that points to an analogue of `tyvar`. The structure of the size and effect grammars (depicted in Figure 4.19) are carefully chosen to allow for unification. The key constraint is that we should never have equivalent values with different representations; otherwise, unification is much more difficult.

For effects, this is enforced by representing integers in unary, using the successor constructor (`FSucc`). For sizes, this is slightly more complicated, since we might want to represent an addition of unknown sizes (e.g. if we concatenate an a-bit integer with a b-bit integer, we get an $(a + b)$-bit integer). We partially resolve the problem of ambiguity by putting lists into a normal form before unification, but this solution is not perfect. It might occur that a program asks us to unify e.g. $a+b$ with $c+d$; our type system cannot represent such a situation, so unification will fail. Fortunately,

[8]As before, types proper are formed by attaching a location to a raw type.

this is an exceedingly rare occurrence – in fact, it has *never* actually come up in practice – and so represents only a minor limitation of the system.

Well-formedness checks In addition to applying Pipeline Types, Lucid also performs several simpler (often purely syntactic) checks to make sure that the user's code "makes sense". Memops are the prime example of this: in addition to the fact that they must typecheck normally, memops are also restricted to just a few syntactic forms, which are verified during this pass. Another check is ensuring that each non-extern event has exactly one handler associated with it. Finally, there are several smaller checks, such as ensuring constructors (which are evaluated at compile-time) don't call runtime functions.

4.4.3 SMT Encoding

As Lucid's type system walks through a function or event handler, it accumulates a collection of *constraints* that describe the conditions under which it is safe to execute that function/handler. As such, being able to test if a collection of constraints is satisfied or satisfiable is crucial. Lucid accomplishes this by encoding the constraints into a decidable fragment of the Theory of Arrays, and using the Z3 SMT solver [27] to evaluate them. This subsection describes that encoding.

Although we run a large number of queries per program (once per function call), each one is typically small enough that we get good performance nonetheless (§4.4.3)

We encode locations using Z3's Array sort, using a strategy inspired by Bradley et al. [15]. Z3 Arrays are essentially infinite integer lists; we embed our (finite) lists into these by setting all unused entries to -1.

Specifically, we encode each location ℓ as a function $\texttt{select}_\ell$ such that $\texttt{select}_\ell(i)$ is the ith element of ℓ. For concrete locations, and those that contain only bounded variables, the encoding is straightforward. For each bounded variable b, we introduce

$$\texttt{select}_\ell(i) = \begin{cases} 0 & i = 0 \\ B_b + 2 & i = 1 \\ 1 & i = 2 \\ -1 & \text{otherwise} \end{cases}$$

(a)

$$\texttt{select}_\ell(i) = \begin{cases} \texttt{select}(A_\alpha, i) & 0 \leq i < L_\alpha - 1 \\ \texttt{select}(A_\alpha, i) + n & i = L_\alpha - 1 \\ \texttt{select}_{\ell'}(i) & L_\alpha \leq i < L_\alpha + \texttt{len}(\ell') \\ -1 & \text{otherwise} \end{cases}$$

(b)

Figure 4.20: `select` functions for ℓ when (a) $\ell = 0.(b+2).1$ and (b) `hd` $\ell = (\alpha + n)$ and `tl` $\ell = \ell'$

a new Int-Sort SMT variable B_b, constrain it to be nonnegative, and return it from the `select` function as necessary. For example, if $\ell = 0.(b+2).1$, we would add a new variable B_b, a new constraint $B_b \geq 0$, and define $\texttt{select}_\ell$ as in Figure 4.20 (a). This is easily represented in SMT as a nested if-then-else expression.

The tricky part is encoding locations that begin with a location variable $\alpha + n$. Since α represents a location, we have to encode it as an Array-sort variable. In fact, we create two new variables: A_α and L_α, where A_α represents α itself and L_α is an Int-sort variable representing the length of A_α.

We then encode our `select` function as follows. First, we define the Z3 expression $\texttt{len}(\ell)$ to be the length of ℓ if ℓ does not begin with a location variable α, and define $\texttt{len}(\ell) = L_\alpha + \texttt{len}(\texttt{tl}\ \ell)$ otherwise. Now assume `hd` $\ell = (\alpha + n)$ and `tl` $\ell = \ell'$. Since αs can only appear at the beginning of a location, we can encode $\texttt{select}_{\ell'}$ as in the earlier paragraph. Using `select` to denote Z3's built-in Array indexing operation, we define $\texttt{select}_\ell$ as in Figure 4.20 (b). We also add constraints that the result of selecting from A_α is always nonnegative, since our location lists never contain negative entries.

Encoding Constraints

Given our location encoding, we encode the constraint $\ell_1 < \ell_2$ as

$$\exists i < \texttt{len}(\ell_1).\, (\texttt{select}_{\ell_1}(i) < \texttt{select}_{\ell_2}(i) \wedge \forall j < i.\texttt{select}_{\ell_1}(j) = \texttt{select}_{\ell_2}(j))$$

Because the existential quantifier appears at the beginning of the constraint, we may remove it via Skolemization, resulting in a query that contains only universal quantifiers. We have found this encoding works quickly without any modifications, but it is possible to remove the universal quantifiers as well, using techniques from Bradley et al. [15]. This shows that the problem is decidable, and empirically has been within the bounds of Z3's capabilities.

Encoding Implication

When typechecking handlers, we need to check whether the user-supplied constraints are sufficient to imply the constraints of the body. This is difficult because, naïvely, the constraint $C_1 \Rightarrow C_2$ is equivalent to $\overline{C_1} \vee C_2$, and introducing negation runs the risk of quantifier alternation rendering our encoding undecidable. Fortunately, there is a simple fix: the negation of the constraint $\ell_1 \leq \ell_2$ is the (positive) constraint $S(\ell_2) \leq \ell_1$. By negating our inequalities before encoding into SMT, we can encode $\overline{C_1} \vee C_2$ solely in terms of positive atoms.

4.4.4 Evaluation

We evaluate our implementation of Pipeline Types on two metrics: **performance** and **usability**.

Module	Description	LoC	Typing time (sec)
Bloom Filter	Probabilistic set of elements.	53	0.26
+Aging	Entries time out	+74	+0.44
Hash table	Deterministic set of elements	25	0.10
+Cuckoo hashing	Contains multiple stages to deal with collisions	+45	+0.22
Hash table w/ timeout	Deterministic set of elements, plus the time each was last touched	65	0.38
+Cuckoo hashing	Contains multiple stages, and clears timed-out entries automatically	+81	+0.31
Bidirectional Map	Stores lists of integers in an array, mapping each to/from its index	39	1.1
Count-min sketch	Probabilistically counts the number of times an element is accessed	70	0.45
+Aging	Entries time out	+83	+0.71

Table 4.1: Modules implemented in Lucid. All make heavy use of polymorphism, records, and vectors. When one module builds on other modules, we indicate the additional lines of code (LoC) with a +. Adding aging to modules was done using a sliding window technique [55].

Performance

To demonstrate the usefulness of Pipeline Types, we reimplemented the example applications presented in the original Lucid paper [73], which were written before Pipeline Types were developed. In the process, we implemented several widely used networking data structures as stand-alone modules (listed in Figure 4.1), each needed by one or more applications. All of these modules utilize polymorphism, records, vectors and abstract types to provide a flexible, reusable, and abstract interface.

We found that on the whole, the example applications benefited substantially from Pipeline Types. Most programs used conventional data structures, which, in the original version of Lucid, had to be inlined into a monolithic application[9], leading to lengthy and obscure code. Once those data structures were defined as independent, reuseable modules, the code became much clearer. In all but one case, the code became much shorter as well; the exception was the Simple NAT application, in

[9]Much like the original Bloom Filter code in this and the previous chapter.

Application	Description	Modules Used	Prev. LoC	New LoC	Typing time (sec)
Stateful Firewall	Blocks unsolicited packets.	Cuckoo Hash w/ Aging	189	37	.68
Closed-loop DNS Defense	Identify/counter DNS reflection attacks	Bloom Filter w/ Aging Cuckoo Hash w/ Aging	215	52	1.8
*Flow [74]	Collects packets by flow for analysis.	Vectors only	149	104	0.03
Distributed Prob. Firewall	Synchronize firewall across multiple switches	Bloom Filter	66	39	0.28
+Aging	Entries in the firewall time out	Bloom Filter /w Aging	119	40	0.75
Simple NAT	Performs network address translation	Bidirectional Map	41	62	1.5
Historical Prob. Queries	Allows queries of frequency for traffic flows	Count-min sketch w/ Aging	93	26	1.2

Table 4.2: Applications implemented in Lucid. Lines of code (LoC) is for the application alone, not including comments or the LoC for the modules on which it depends (see Table 4.1 for the latter).

which the boilerplate of defining a NAT-specific module was significant compared to the original program size. A list of these programs appears in Figure 4.2.

The Lucid paper also reported three other applications (simple chain replication, single-destination RIP, and automatic rerouting), but they were either very simple or highly specialized for their particular task. We do not report on them here because they saw essentially no change from Pipeline Types.

For both the modules and the full applications, we found that typechecking times were low, with even the longest example taking under 2 seconds. This is good; it means that users are not paying a significant price for the power of Pipeline Types.

4.4.5 Usability

When implementing Pipeline Types, we did so with the goal of making them as easy-to-use as possible. We approached this goal from two directions: making sure the user got *useful feedback*, and ensuring the system did not make writing programs *harder* by modifying the language's syntax as little as possible.

Error Messages

Our system of Pipeline Types provides programmers with a simple, high-level directive to follow in their programs: *global values must be used in the order they are declared*. This directive is valuable in its own right, for it provides an easy guideline on how to write correct programs. Moreover, the guideline also allows us to provide concise, but extremely useful error messages.

```
In tmp/Errors/ordering_error2/ordering_error.dpt:

8|    Array.set(reg1, 0, tmp);
      ~~~~~~~~~~~~~~~~~~~~~~~~

error: tmp/Errors/ordering_error2/ordering_error.dpt: Function call violates the
global order
```

Figure 4.21: The output of the Lucid compiler when given a program that contains an ordering error.

A sample error message is shown in 4.21. On its surface, the message may not appear to contain much information; it merely informs the user that there was an ordering error, and points to the line where it was detected. However, this line number is *extremely useful*, since with that information the user need only scan backwards in the program to find the last global access. With that information, they can see exactly why the error occurred (the globals will necessarily have been used out-of-order), and can begin figuring out how to solve the problem.

Syntax Changes

Our implementation of Pipeline Types is carefully designed to have a minimal impact on the surface language. Thanks to type inference, we are able to automatically determine almost everything the type system needs. The only exceptions are constraints in module interfaces and on events with global arguments, which are difficult or impossible to infer.

As a result, the addition of Pipeline Types requires users to manually write those constraints. However, doing so is not onerous; our constraint syntax is carefully designed to make no reference to the underlying type system, instead referring to high level concepts such as `start` and `end`, and using variable names instead of location annotations. Furthermore, the need to write constraints in the first place is not an everyday occurrence: constraints in module interfaces need only be written once, while events with global arguments are extremely rare in practice.

In short, the power of pipeline types imposes only minimal burden on programmers, while providing them with a powerful guideline for writing correct programs and useful corrections should they fail to do so. This is a small price to pay to eliminate one of the most frustrating classes of error in modern dataplane programming.

4.5 Related Work

The past decade has seen the introduction of several new dataplane languages, including Domino [70], Chipmunk [32], Lyra [31], and P4All [36]. Like Lucid, these languages offer higher-level abstractions than P4; unlike Lucid, they then typically rely on synthesis techniques (often SMT-based) to actually lay out the program in a switch. For example, Lyra attempts to split a single program across multiple switches, while P4All solves an Integer Linear Programming problem to find optimal sizes for each data structure.

The common weakness of these synthesis techniques is that, if they fail, they do so with next to no useful feedback. If a program contains an ordering error, an SMT solver will conclude that it cannot be laid out along a pipeline, but it will not tell the user *why*. This is a significant issue, especially as programs get larger and more complex! Pipeline types allow Lucid to "screen out" many common errors, making those compilation errors less mysterious, and easy to debug. Pipeline types need not be restricted to Lucid, either; we hope that other dataplane languages will take inspiration and include a similar system.

Effect systems Outside of the domain of networking, type-and-effect systems have been used to control memory access since the 80s [34] and grew to prominence in the 90s with the work of Tofte, Talpin, Birkedal and others on region inference [77, 76]. These systems protected against use-after-free errors, but did not constrain access order along a pipeline as Lucid does. Later, researchers developed type systems for specifying more general "resource usage protocols" [28, 40]. Such systems can specify constraints on the order in which resources are used, but the protocols involved have a different character (often characterized by regular languages rather than numeric, ordered, hierarchical locations), use different technical machinery, and were targeted at different applications.

An alternative to type-and-effect systems are those type systems based on linear [35] or ordered logic [60]. Ordered type systems generate similar kinds of constraints to Pipeline Types, effectively constraining the order in which data is accessed, but they have not been applied to packet processing pipelines. Moreover, to be effective they would likely need to be enriched with a variety of new features such as hierarchical locations, ordering constraints and new rules for managing vectors and loops.

Chapter 5

Compiling Lucid

Lucid is designed to be a practical language for writing programs that are run in real networks. This is accomplished by compiling Lucid programs to P4, which is then further compiled to switch hardware. In particular, the Lucid compiler produces P4 code that is specialized for compilation to the Intel Tofino[1]. We chose to target the Tofino because it is a commonly used programmable switch for networking researchers, including those at Princeton. In the future, we hope to extend our compiler to other targets, such as eBPF, FPGAs, or SmartNICs.

This chapter describes the compilation process from Lucid to Tofino-specialized P4. The remainder of the compilation (from P4 to a Tofino binary) is handled using Intel's proprietary P4-to-Tofino compiler. However, the Lucid compiler takes advantage of Lucid's higher-level code to pre-emptively perform several compilation tasks that would normally be performed by the Tofino compiler, for example by manually scheduling statements in match-action tables. In practice, we have found that doing these tasks at the Lucid level produces superior results; in particular, we can compile much larger programs, and do so faster, than when using the Tofino compiler (§5.7).

[1]For example, the P4 code is structured as a series of match-action tables that are designed to fit within the Tofino's hardware, and are annotated with the stage we expect them to be placed in.

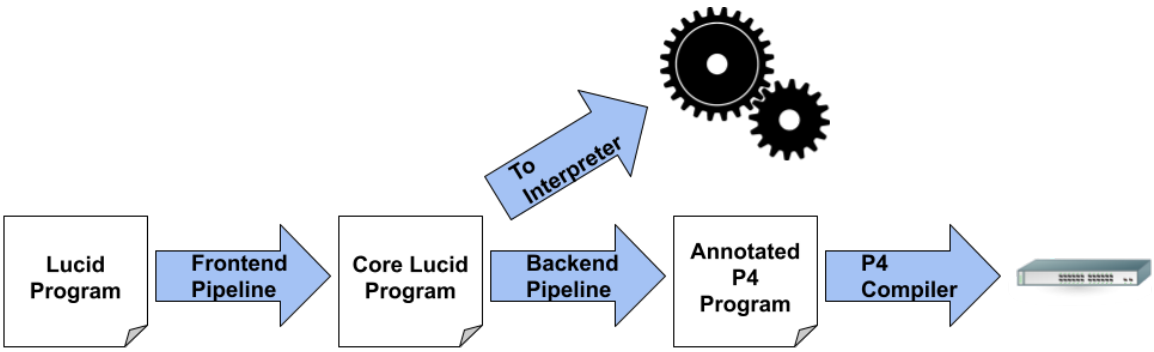

Figure 5.1: A high-level overview of Lucid's system architecture.

5.1 Compiler Overview

The high-level architecture of this system is illustrated in Figure 5.1. As shown, the Lucid-to-P4 compilation process can be conceptually split into two stages. The **Frontend Pipeline** simplifies the syntax of Lucid, removing higher-level language constructs to produce a program written in a minimal subset of Lucid that we call **Core Lucid**. The **Backend Pipeline** then takes this simplified program and incrementally transforms it into a P4 program.

The compilation process is structured as a series of source-to-source transformations of the Lucid/Core Lucid program. Since Core Lucid is a subset of Lucid, the program remains valid Lucid code until the very final step, when the resulting P4 code is emitted. As a result, we can re-use any Lucid-based infrastructure such as typechecking or pretty-printing (for debugging purposes) at any point in the compilation process. This also demonstrates the generality of Lucid – we do not need a specialized IR when compiling to P4, just Lucid itself.

Aside: Frontend vs. Backend The distinction between the Frontend and Backend pipelines follows the standard definitions for compilers: the Frontend applies target-independent simplifications to the source code, and the Backend applies target-specific transformations that specialize the code for the desired hardware. If in the future we add the ability to compile to targets besides the Tofino, we will likely re-

use the same Frontend passes, but will only be able to re-use some (or none) of the existing Backend passes.

5.1.1 Core Lucid

Core Lucid is a subset of Lucid that we use as an intermediate representation; the goal of the frontend pipeline is to produce a Core Lucid program. Specifically, Core Lucid programs have the following key differences from full Lucid programs:

- **No user-defined modules**: All programs are "flat".
- **No user-defined functions**: All such functions are inlined.
- **No events with global arguments**: All such events are replaced with copies that do not take global-typed arguments.
- **No non-global compound types**: All records, vectors and tuples are replaced with collections of independent variables.

5.1.2 Attribution

The Frontend Pipeline was developed primarily by the author, with support from John Sonchack and David Walker. The Backend Pipeline was developed primarily by John Sonchack; The author's contributions were in discussing high-level ideas for passes and reviewing code. The original Lucid paper [73] described an initial version of this compilation process at a high level.

5.2 Frontend Pipeline

The goal of the Frontend Pipeline is to produce a Core Lucid program by eliminating high-level language features. This section describes the major transformations that we

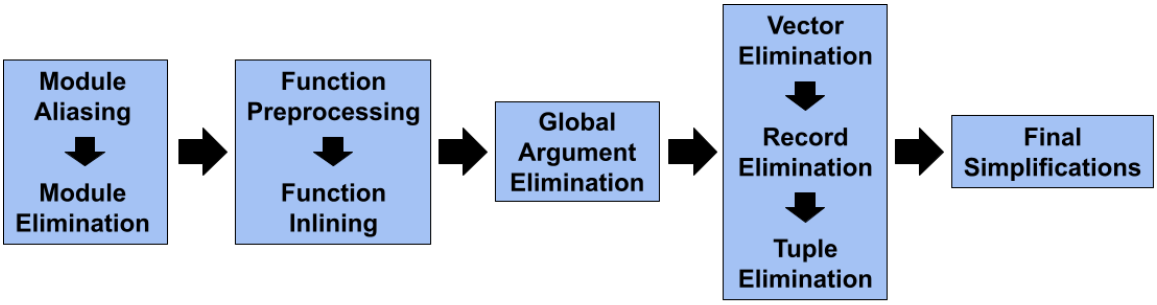

Figure 5.2: A visualization of the major transformations done in the Frontend Pipeline, grouped by purpose.

employ to do so. In addition, we employ several common compiler transformations, such as inlining constant variables and alpha-renaming to ensure all variable names are unique. Because these transformations are utterly standard, we will ignore them in this section except for noting when they are required.

Figure 5.2 shows the high-level structure of the Frontend Pipeline. We have grouped the major transformations according to their purpose, which mirrors the structure of this section.

5.2.1 Unpacking Modules

Modules are a powerful tool for building modular programs, but the introduction of multiple namespaces makes syntax-directed transformations painful. One of the first goals of the frontend is to eliminate modules, creating a single flat program that is easier to manipulate. We do so in two steps.

Module Aliasing The first step is to remove module alias statements like `module M = M1 if b else M2`. We do so in a straightforward way: we evaluate `b`, and substitute `M` with either `M1` or `M2` throughout the program depending on the result. We expect `b` to be a literal, though it may have originally been a variable that was inlined by a previous pass.

Module Elimination The next step is to get rid of module declarations by hoisting the code inside the module to the top level, as if it had been written non-modularly in the first place. A side effect of this is to remove the module's interface, but this is fine; the abstraction benefits of the interface were already enforced during typechecking. More problematic is the loss of the module's namespace; the rest of the program still refers to its contents as if they were inside the module, so we must adjust such references to target the new top-level definitions instead. This process is demonstrated in Figure 5.3.

```
1  module BF {
2    type filter = ...;
3
4    constructor filter create(...);
5
6    fun add(filter f, int elem)
7      { ... }
8  }
9
10 global BF.filter f =
11   BF.create(...);
12
13 event e(int elem) {
14   BF.add(f, elem);
15 }
```

(a)

```
1
2  type BF_filter = ...;
3
4  constructor BF_filter create(...);
5
6  fun BF_add(BF_filter f, int elem)
7    { ... }
8
9
10 global BF_filter f =
11   BF_create(...);
12
13 event e(int elem) {
14   BF_add(f, elem);
15 }
```

(b)

Figure 5.3: Module Elimination. (a) shows a program using a simple Bloom filter module `BF`. (b) shows an equivalent program in which the module is eliminated.

5.2.2 Function Inlining

PISA hardware does not support general-purpose function calls; therefore, we inline all function definitions during compilation. This process has two steps: Function Preprocessing and Function Inlining.

Function Preprocessing When we inline functions, we replace their return statements with assignments to a designated return variable. This changes the semantics of the function, since execution stops at return statements but not at assignments.

Accordingly, the Function Preprocessing step reorganizes each function's body so that no executable code follows any return statement, using a technique known as if-conversion [6].

To illustrate our strategy, consider the sequenced statements `s1; s2` in a function body. We first examine `s1` to determine if it returns along all control paths, no control paths, or some (but not all) control paths. If `s1` never returns, then we needn't modify it, and simply move on to `s2`. If `s1` always returns, then `s2` is dead code and may be eliminated. The tricky part is if `s1` only sometimes returns.

In this case, we introduce a boolean variable that tracks whether the function has reached a return statement, and wrap `s2` in an `if`-statement so it is only executed if that variable is `false`. We ensure the variable is set to `false` at the beginning of the function, and is set to `true` immediately before each return statement. This translation is illustrated in Figure 5.4; we would translate the function `f` in Figure 5.4a to `f_transformed` in Figure 5.4b.

In some cases, we may perform optimizations to reduce the impact of the transformation. For example, if we have an `if` statement with exactly one branch guaranteed to return, we may safely move the remaining code into the other branch without introducing an extra variable or `if` statement. This technique is demonstrated in Figure 5.4c.

Function Inlining The next step is to inline the modified function definitions. We do so in a fairly standard way; when we have a statement containing a function call, we insert the function body immediately beforehand, with the parameters replaced with the corresponding arguments. We then create a new variable to hold the function's return value, replace all return statements with assignments to that variable, and replace the call itself with the return variable, as shown in Figure 5.5.

One caveat is that function bodies may mutate their parameters, as in Figure

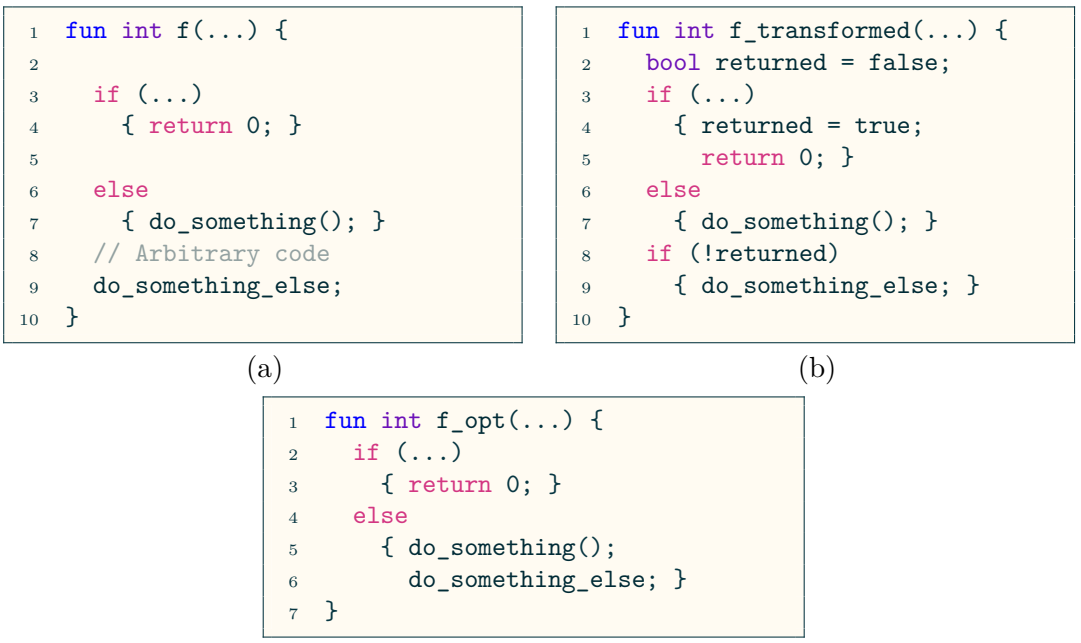

Figure 5.4: Function Preprocessing: (a) shows a simple function definition. (b) shows how to transform it so that no executable code ever follows a return statement. (c) shows an alternative transformation that does not introduce an extra variable. Note that the code added in (b) does nothing at this stage of compilation; it will be useful later, when return statements are converted to variable assignments.

5.6a. When we subsitute arguments for parameters, we need to ensure that we are not accidentally mutating an existing variable. We do so by creating new variables for any parameters that are mutated, as demonstrated in Figure 5.6b.

5.2.3 Eliminating Global Event Arguments

Lucid allows users to define events with global-typed parameters, allowing one to write an event that e.g. updates a different array depending on its arguments. Unfortunately, the hardware does not support such dynamic updates: the target of each update must be known statically, at compile time. Functions with global arguments do not pose a problem, as they will be inlined; however, handlers cannot be inlined in general. Instead, we duplicate each event with global arguments so that there is one copy of the event for each possible argument (or combination of arguments). This is

```
1  fun int f(int x, int y) {
2    if (x < y)
3      { return 0; }
4    else
5      { do_something();
6        return x + y; }
7  }
8
9  event e(int n, int m) {
10   generate bar(f(n, m));
11 }
```

(a)

```
1  event e(int n, int m) {
2    int retval = 0;
3    if (n < m)
4      { retval = 0; }
5    else
6      { do_something();
7        retval = n + m; }
8    generate bar(retval);
9  }
```

(b)

Figure 5.5: Function Inlining: (a) shows a simple function that does not mutate its arguments. (b) shows how that function can be inlined into the body of event `e` by replacing return statements with assignments to a designated variable.

```
1  fun int f(int x, int y) {
2    if (x < y)
3      { x = y; }
4    else
5      { do_something(); }
6    return x + y;
7  }
8
9  event e(int n, int m) {
10   generate bar(n + f(n, m));
11 }
```

(a)

```
1  event e(int n, int m) {
2    int retval = 0;
3    int x_arg = n;
4    if (x_arg < m)
5      { x_arg = m; }
6    else
7      { do_something(); }
8    retval = x_arg + m;
9    generate bar(n + retval);
10 }
```

(b)

Figure 5.6: Function Inlining: (a) shows a function that mutates its argument. (b) shows how to inline this function into the body of event `e`, by creating a new variable representing the mutated function argument.

demonstrated in Figure 5.7; notice that we do not generate a copy for `arr2`, since it had the wrong type.

In rare cases, events may take multiple global arguments. In these cases, they must also supply constraints about the relative order of those arguments. Figure 5.8a shows an event with two arguments, whose first argument must be earlier in the global order than its second. These constraints are considered when duplicating events; Figure 5.8b contains only the copies of `foo` that have the appropriate ordering.

```
global Array.t<32> arr1 = ...;
global Array.t<16> arr2 = ...;
global Array.t<32> arr3 = ...;

event foo(Array.t<32> a,
          int idx,
          int val) {
  Array.set(a, idx, val);
}
```

(a)

```
global Array.t<32> arr1 = ...;
global Array.t<16> arr2 = ...;
global Array.t<32> arr3 = ...;

event foo_arr1(int idx, int val) {
  Array.set(arr1, idx, val);
}

event foo_arr3(int idx, int val) {
  Array.set(arr3, idx, val);
}
```

(b)

Figure 5.7: Global Argument Elimination: (a) shows an event `foo` that takes a global-typed argument. (b) shows how to eliminate `foo` by replacing it with two events that have the global arguments baked in. No copy was created for `arr2` because it has the wrong type.

5.2.4 Eliminating Non-Global Compound Types

Lucid provides three non-global compound types: records, vectors, and tuples. We eliminate all of these constructs by unpacking them, so that each entry is a completely independent variable. Dissociating the entries in this way provides opportunities for optimization later in the compilation process, since each entry can be treated individually. Note that we do *not* eliminate arrays, because they are a hardware primitive and hence cannot be broken down.

Our elimination pass operates by first translating records and vectors into tuples, then unpacking those tuples. By this point in the program, functions and sizes have been inlined, so we know the length of each vector, and don't need to worry about returning compound values from functions.

Vector and Record Elimination Eliminating vectors is straightforward. Since we know the length of each vector, we simply replace each one with a tuple of equal length. At the same time, we unroll all loops in the program, so each indexing operation into the vector has a static index. Thus we can simply replace each operation with a tuple projection at that index.

```
global Array.t<32> arr1 = ...;
global Array.t<32> arr2 = ...;
global Array.t<32> arr3 = ...;

event foo(Array.t<32> a1,
          Array.t<32> a2,
          int idx,
          int val)
          [a1 < a2] {
  Array.set(a1, idx, val);
  Array.set(a2, idx, val);
}
```

(a)

```
global Array.t<32> arr1 = ...;
global Array.t<32> arr2 = ...;
global Array.t<32> arr3 = ...;

event foo_arr1_arr2(int idx, int
    val) {
  Array.set(arr1, idx, val);
  Array.set(arr2, idx, val);
}

event foo_arr1_arr3(int idx, int
    val) {
  Array.set(arr1, idx, val);
  Array.set(arr3, idx, val);
}

event foo_arr2_arr3(int idx, int
    val) {
  Array.set(arr2, idx, val);
  Array.set(arr3, idx, val);
}
```

(b)

Figure 5.8: Global Argument Elimination: (a) shows an event `foo` that takes two global-typed arguments, with the constraint that the first must precede the second in the global ordering. (b) shows how to eliminate this event, by creating one copy for each pair of valid arguments.

Record elimination is similar to vector elimination, but even simpler since it does not involve loops or sizes. We replace each record type with a tuple, whose entries store the fields of the record in declaration order. We replace record projection operations with tuple projection at the appropriate index.

Vector and Tuple elimination are both demonstrated in Figure 5.9. The programs in Figures 5.9a and 5.9b both produce the program in Figure 5.9c after vector/record elimination, respectively.

Tuple Elimination Now all compound values in the program are represented as tuples, so we may unpack all of them at once. To do so, we split each tuple into a series of independent variables, one for each entry. Tuple assignments then become a series of independent assignments, and we can replace each tuple projection with the variable representing the projected index. In the case of nested tuples, we unbox

```
int[3] nums =
    [f(i) for i = 0 to 2]


int acc = 0;
for (i < 3) {
  acc += nums[i]
}
```

(a)

```
type triple =
  { int fst; int snd; int thrd; }

triple nums =
  { fst = f(0); snd = f(1);
    thrd = f(2); }

int acc = 0;
acc += nums.fst;
acc += nums.snd;
acc += nums.thrd;
```

(b)

```
(int * int * int) nums =
    (f(0), f(1), f(2))

int acc = 0;
acc += nums[0];
acc += nums[1];
acc += nums[2];
```

(c)

Figure 5.9: Vector and Record Elimination: (a) and (b) both show equivalent programs using vectors and records, respectively. Both programs compile to (c), which uses only tuples.

them recursively, essentially flattening the tuple. If there are any events that take tuples as arguments, we adjust the event's footprint so that it takes each element of the tuple as a separate argument. Tuple elimination is demonstrated in Figure 5.10.

```
(int * int * int) nums =
    (f(0), f(1), f(2))

int acc = 0;
acc += nums[0];
acc += nums[1];
acc += nums[2];
```

(a)

```
int nums_0 = f(0);
int nums_1 = f(1);
int nums_2 = f(2);

int acc = 0;
acc += nums_0;
acc += nums_1;
acc += nums_2;
```

(b)

Figure 5.10: Tuple Elimination: (a) shows a program implemented using a tuple. (b) shows how to transform that program to be written without tuples.

5.2.5 Final Simplifications

The series of transformations we have described so far are powerful, but most come at the cost of introducing unnecessary code. For example, unpacking tuples can create extra assignments when one field of the tuple is modified while others aren't. Alternatively, inlining a function with an `if` statement in its body may lead to one branch of the `if` always being taken, so the other branch is dead code. We attempt to mitigate these inefficiencies through a combination of constant folding, variable inlining, and dead code elimination, which we refer to collectively as the **Final Simplifications**.

Since all of these are standard techniques, we will not describe them in detail, instead opting to give a high-level summary. In short, we begin by replacing each use of each variable with its definition, if doing so will not change the semantics of the program[2]. Then, we precompute each expression as much as possible, using normal arithmetic and boolean identities to simplify those we cannot compute statically. Finally, we remove any `if` or `match` statements that always take a specific branch, as well as any variables that are no longer used (because all their uses were inlined). The result is, ideally, a program with many fewer variables, and no dead or useless code.

Downsides of the Final Simplifications Unfortunately, variable inlining is a dangeous transformation, since it replaces each use of a variable with its body, potentially increasing the size of the code. One might worry that inlining too many variables might make the program more difficult to compile, especially if we end up duplicating large expressions as a result.

We have found that generally, this does not happen. In practice, the primary bottleneck for compiling to switch hardware has been the number of variables, not the number or size of expressions. This is because large numbers of variables impose high

[2]i.e. the variable is neither the result of a stateful computation, nor depends on other variables that have changed since it was defined.

Packet Header Vector pressure (§2.2.4). In contrast, switches can fit large amount of compute into a single stage so long as there are no dependencies between the computations. Since multiple copies of an expression are necessarily independent of each other, duplicating expressions does not generally cause problems.

Nonetheless, we provide users with the option to mark variable declarations or assignments that should not be inlined, allowing them to reap the benefits of the Final Simplifications, while selectively avoiding the downsides if necessary.

5.3 Backend Compilation Strategy

The result of the Frontend Pipeline is a Core Lucid program, which is passed as input to the Backend Pipeline. Like before, the Backend Pipeline is structured as a series of source-to-source program transformations. However, rather than aiming to simplify the program's syntax, the Backend Pipeline instead aims to produce a P4 program that can be easily compiled to the Tofino. Accordingly, the passes of the Backend Pipeline are each dedicated to changing the structure of the Core Lucid program to match the restrictions of the hardware.

The Backend Pipeline is not guaranteed to produce a compiling program, but it reduces the potential problems greatly, and provides feedback to the user should compilation ultimately fail. Most Lucid programs compile successfully with minor tweaks. We evaluate the compiler in greater detail in §5.7.

This section describes the overall strategy and goals of the Backend Pipeline; the next section describes the actual passes in more detail.

5.3.1 Match-action tables

The basic building blocks of our compiled P4 code are **match-action tables**. Each of these has essentially the same semantics as a `match` statement in Lucid: it matches

one or more values against one or more patterns, and executes the code block (an **action**) corresponding to the first matching pattern, or a default action if no pattern matches. An action consists of a set of statements; when an action is invoked, all of its statements are executed in parallel in the same stage[3]. Each table is located at a particular stage of the pipeline; multiple tables can be invoked in the same stage, in which case all of their chosen actions are executed simultaneously.

There are two kinds of match-action tables. **Static** tables have their patterns and actions (their **rules**) fixed at compile time, while **dynamic** tables may be modified by the control plane while the program is running, adding or removing rules on the fly. Both types of match-action table are represented in Lucid: `match` statements represent static tables, while Lucid Tables (`table`-type globals) represent dynamic tables. The compiler is free to tinker with static tables (e.g. merging tables, matching on additional variables) during compilation, but dynamic tables are visible to the control plane, and so they must present the same interface (i.e. match on the same set of variables) as in the source program.

The fundamental strategy of the Backend Pipeline is to combine our Lucid handlers into a single list of match-action tables (both static and dynamic), which are each invoked sequentially. In Lucid terms, our output program should therefore consist of a series of `match` and `table_match` statements. However, there are further restrictions of the form of these statements.

Atomicity

The fact that all statements in each action are executed simultaneously has two important implications for the rule bodies of our final `match` statements. In particular, it implies that:

1. All statements in the body of each rule must be independent of each other, and

[3]This behavior is specific to the Tofino; it is not part of the general P4 specification.

2. All statements in the body of each rule must be **atomic**, meaning that they can each be executed in a single pipeline stage.

The capabilities of a single pipeline stage are fairly restricted: the fundamental operations they can perform are binary operations on variables and constants, hash statements, and memory (array) accesses. More specifically, we define atomicity as follows:

1. An expression is **immediate** if it is either a variable or a literal.
2. The following expressions are atomic:
   (a) An immediate expression.
   (b) A binary operation whose operands are immediate.
   (c) A hash expression whose arguments are immediate.
   (d) A function call (such as to `Array.update_complex`) whose arguments are immediate, *except* calls to `table_match`.
3. The following statements are atomic:
   (a) Variable definitions and assignments whose body is an atomic expression.
   (b) Generate statements whose body is an atomic call to an event constructor.
   (c) Table install statements (`table_install`) whose arguments are immediate.

Top-level matches

With this definition, we can define the form of our top-level match statements. Specifically, our output program should be a series of `match` and `table_match` statements, such that, for each `match` statement,

1. Each expression it matches on is immediate.

2. Each statement in the body of each rule is atomic[4].

3. Each statement in the body of each rule is independent of the other statements in that rule's body.

In addition, we allow `table_match` statements to be wrapped in simple conditionals; for simplicity, we defer discussion of this detail to §5.4.3.

Benefits of match-action tables

Match-action tables are the key to efficiently fitting our programs into the hardware. One of their greatest strengths is the ability to compare large numbers of variables in parallel. For example, consider the code in Figure 5.11a.

```
if((x == 4 and y == 17)
    or z == 42) {
    do_something();
} else {
    do_something_else();
}
```

(a)

```
// Stage 1
bool cmp_x = (x == 4);
bool cmp_y = (y == 17);
bool cmp_z = (z == 42);
// Stage 2
bool and_x_y = cmp_x and cmp_y;
// Stage 3
bool b = and_x_y or cmp_z;
// Stage 4
if(b) {
    do_something();
} else {
    do_something_else();
}
```

(b)

```
match x, y, z with
| 4, 17, _ -> { do_something(); }
| _, _, 42 -> { do_something(); }
| _, _, _  -> { do_something_else(); }
```

(c)

Figure 5.11: Testing Equality via Match: (a) shows a program that tests three boolean values to determine what to do next. (b) shows an inefficient compilation strategy that evaluates each condition using ALUs. (c) shows a much more efficient strategy that uses match statements instead. The inefficient code in (b) uses 4 stages, while the strategy in (c) needs only one.

[4]Note that this implies that no rule body contains a `table_match`; all `table_match`es must happen at top level.

Recall that on the hardware, each statement has to be atomic; for arithmetic and boolean operations, the atomic form is a binary operation. Thus if we want to naïvely compute the condition for the `if` statement in Figure 5.11a, we must split it up into 5 separate computations. Due to dependencies between those computations, we must spread those computations across the pipeline, ultimately taking 3 stages to compute the condition before finally testing against it in the fourth (Figure 5.11b). In contrast, using a `match` statement (i.e. a static match-action table), we can compare all three variables at once, while encoding the boolean operators in the structure of the rules, executing the statement in only a single stage (Figure 5.11c).

5.3.2 Well-formedness

The Lucid compiler attempts to automatically enforce as many hardware constraints as it can. However, not all restrictions can be enforced this way. As a result, our compiler requires the following well-formedness conditions of programs that are compiled to the Tofino; these conditions do not apply to Lucid programs targeting other backends (such as the interpreter, or a future compiler to different hardware). Programs failing any of these conditions will be rejected by the compiler.

Specifically, we require the following:

1. Each parameter to each event must either begin or end on a byte boundary; the first argument is assumed to begin on a byte boundary. This is necessary to ensure the arguments fit neatly into the Packet Header Vectors (PHVs); arguments that span multiple PHVs are much harder to work with.

2. Each array must fit within a certain amount of memory (the maximum amount of memory that can be allocated to a single RegisterArray on the Tofino).

3. On each control flow path, each handler may generate at most one event at a

network location[5] other than the current switch.

The final condition bears further explanation. When compiled to the Tofino, `generate` statements do not immediately create a new packet; rather, they store that event's arguments in reserved header fields of the current packet. When the current packet reaches the end of the pipeline, it is cloned, with one copy being made for each event that was generated during processing. The arguments of those events are each attached to the corresponding copy, which is then forwarded to the appropriate destination.

The Tofino hardware only allows packets to be sent to a single network location, plus possibly the recirculation port. It also cannot send multiple packets out of the same port at once. Hence we cannot e.g. send event A out of port 1 and event B out of port 2; similarly, we could not send both events A and B out of port 1. This gives rise to restriction (3) above.

5.3.3 Output form

The goal of the Backend Pipeline is to produce a program that can be easily compiled to the Tofino. In particular, we wish to ensure that:

1. There is only one event, which combines the arguments and handler bodies of all the user-defined events.

2. That event's handler is a series of top-level statements (as defined in §5.3.1).

3. Each array access is performed using `Array.update_complex`.

4. For each array, every `Array.update_complex` call uses the same variables as arguments.

[5] That is, a port, a switch identifier, or a multicast group.

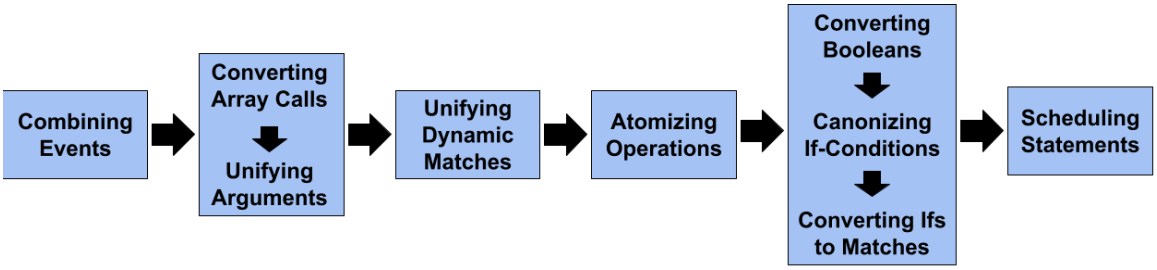

Figure 5.12: A visualization of the major transformations done in the Backend Pipeline, grouped by purpose.

Item (1) reflects the fact that ultimately, all packets are processed by the same pipeline, which must simultaneously implement all event handlers. Item (2) ensures that we can translate each statement directly to a match-action table. Item (3) ensures that we can translates array accesses directly to P4 RegisterActions (which are directly represented in Lucid by `Array.update_complex`). Item (4) reflects an underlying hardware constraint that each access to a given array draws its arguments from the same positions in the PHV; unifying each argument into a single Lucid variable enforces this.

5.4 Backend Pipeline

The structure of the Backend Pipeline is depicted in Figure 5.12. Conceptually, it can be thought of as a series of transformations to put the program in the form described in §5.3.3, followed by a scheduling algorithm to ensure that the program fits into the Tofino's hardware resources.

The major steps in this process are:

1. Combining all events and handlers into a single event and handler (§5.4.1).

2. Converting all array accesses to use `Array.update_complex`, and unifying their arguments (§5.4.2).

3. Unifying all the `table_match` statements for each dynamic table, and placing

them at top level (§5.4.3).

4. Breaking down statements into their atomic components (§5.4.4)

5. Transforming all boolean computation to happen using `match` statements (§5.4.5).

6. Scheduling each statement into `match` statements that can be directly mapped to Tofino match-action tables (§5.5).

5.4.1 Combining Events

A typical Lucid program consists of many events and handlers that execute on different "types" of packet. However, the resulting P4 program only has one block of code which processes *all* packets. We account for this by combining all the events and handlers in the Lucid program into a single event representing an arbitrary incoming packet. Our strategy for doing so is straightforward: we create a new event whose arguments are a tagged union of the arguments of each other event, and whose handler matches on the tag and then executes the code of the corresponding event's handler. This transformation is illustrated in Figure 5.13.

```
1 event foo(int arg1, int arg2) {
2     foo_stuff(arg1, arg2);
3 }
4
5 event bar(int arg3, int arg4) {
6     bar_stuff(arg3, arg4);
7 }
```

(a)

```
1 event main(int tag,
2             int arg1, int arg2,
3             int arg3, int arg4) {
4   match tag with
5   | 0 -> foo_stuff(arg1, arg2);
6   | 1 -> bar_stuff(arg3, arg4);
7 }
```

(b)

Figure 5.13: Unifying Events: (a) shows a program with two events. (b) shows how to combine those events into one by creating a tagged union of their arguments.

After the transformation, the Lucid parser is responsible for ensuring that the tag is set appropriately for each incoming packet. User-written parsers are automatically adjusted to do so. Notice that if the tag is somehow set to an unexpected value, no code will be executed, and the packet will be dropped by default.

5.4.2 Converting Memory Accesses

Lucid's `Array` abstraction hides much of the complexity of dealing with stateful memory (i.e., registers) in a P4 program. The chief example is the ordered type system, but Lucid also hides additional constraints. First, the only memory access operation supported by the hardware corresponds to `Array.update_complex`, so we begin by converting all array accesses to this form. Second, the local values passed to each operation on the same array must always come from the same locations in the underlying packet's header; in Lucid terms, this means that the arguments must always be stored in the same variable[6].

Converting Array Calls

Our first goal is to convert each array access into a call to `Array.update_complex`. We do so in two parts. First, we convert every call (that is not already to `Array.update_complex`) into a call to `Array.update`. This translation is outlined in Figure 5.14, using `id` and `write` to represent memops that always return their first and second argument (i.e., the value in memory or the local value), respectively[7].

```
1 Array.get(arr, idx);
2
3 Array.getm(arr, idx,
4             getop, localv);
5 Array.set(arr, idx,
6           localv);
7 Array.setm(arr, idx,
8             setop, localv);
```

(a)

```
1 Array.update(arr, idx,
2              id, 0, id, 0);
3 Array.update(arr, idx,
4              getop, localv, id, 0);
5 Array.update(arr, idx,
6              id, 0, write, localv);
7 Array.update(arr, idx,
8              id, 0, setop, localv);
```

(b)

Figure 5.14: Converting Arrays: (a) shows calls to each of the basic `Array` functions. (b) shows how those calls can be translated into calls to `Array.update` using the primitive memops `id` and `write`.

Once we've done this, we need only convert `Array.update` calls. Doing so re-

[6]Or multiple variables that can be overlaid in the P4 program, but since we cannot control which variables the Tofino compiler chooses to overlay we ignore this possibility.

[7]For a refresher on memops, see §3.2.6.

quires us to combine two small (2-argument) memops to create a 3-argument complex memop. For each pair of memop arguments to `Array.update`, we combine them as depicted in Figure 5.15, using `cell1` to evaluate the memop that writes to memory, and `cell2` for the one that returns to the handler.

```

memop write_op(int memval1,
               int localval1) {
  if(<write_cond>) {
    return <write_e1>;
  } else {
    return <write_e2>;
  }
}

memop read_op(int memval1,
              int localval2) {
  if(<read_cond>) {
    return <read_e1>;
  } else {
    return <read_e2>;
  }
}

Array.update(arr, idx,
             read_op, v1,
             write_op, v2);
```

(a)

```
memop read_write(int memval1,
                 int localval1,
                 int localval2) {
  bool b1 = <write_cond>;
  bool b2 = <read_cond>;

  // Value of cell1 is written back
  // to memory
  if (b1)  { cell1 = <write_e1>; }
  else {
  if (!b1) { cell1 = <write_e2>; } }

  // Value of cell2 will be returned
  // to the handler
  if (b2)  { cell2 = <read_e1>; }
  else {
  if (!b2) { cell2 = <read_e2>; }

  if (true) { return cell2; }
}

Array.update_complex(arr, idx,
                     read_write,
                     v1, v2);
```

(b)

Figure 5.15: Combining Memops: (a) shows two 2-arguments memops. (b) shows how they can be combined into a single 3-argument memop.

Unifying Array Arguments

Now that all Array accesses have the same form, we must ensure that each access to each array uses the same variables for the "local value" arguments (the additional arguments to each memop). Our process is demonstrated in Figure 5.16. We begin by computing, for each array, the sets of variables used for each argument. If those sets are singletons (or empty, if all arguments were constant), then we are done. Otherwise, for each set with at least two elements, we create a fresh variable. We

then insert assignments to that variable before each update operation, and use the variable as the local value argument to the update.

```
1
2  int v1 = x;
3  int v2 = y;
4
5  if(...) {
6
7    Array.getm(arr, idx, op, v1);
8  } else {
9
10   Array.getm(arr, idx, op, v2);
11 }
```

(a)

```
1  int v  = 0;
2  int v1 = x;
3  int v2 = y;
4
5  if(...) {
6    v = v1;
7    Array.getm(arr, idx, op, v);
8  } else {
9    v = v2;
10   Array.getm(arr, idx, op, v);
11 }
```

(b)

Figure 5.16: Unifying Array Arguments: (a) shows a program that accesses an array using two different local variable arguments. (b) shows how to transform the program so that the same argument is always used. We use `Array.getm` here for illustrative purposes; in practice it would have been transformed into `Array.update_complex` by this point.

This approach is slightly inefficient, since it introduces a new variable assignment that might increase the number of stages used. If the arguments for a particular local value are never alive at the same time, we can avoid this overhead by directly overlaying those arguments, storing them in the new variable from the get-go. This tactic is demonstrated in Figure 5.17; Figure 5.17b shows the naïve translation, while Figure 5.17c shows the optimized version.

5.4.3 Dynamic tables

Recall that Lucid Tables are a direct representation of dynamic match-action tables on the hardware. Unlike other types of statements, the hardware does not permit nested table matches. Hence any access to a Lucid Table (i.e. a `table_match` statement) must occur at top level, rather than inside a `match` statement like everything else.

This poses two challenges. First, a given Lucid Table may be matched along several different control paths. However, as global objects, Lucid Tables may be

```
1
2  if(...) {
3    int v1 = x;
4    // Do stuff with v1
5
6    Array.getm(arr, idx, op, v1);
7  } else {
8    int v2 = y;
9    // Do stuff with v2
10
11   Array.getm(arr, idx, op, v2);
12 }
```

(a)

```
1  int v = 0;
2  if(...) {
3    int v1 = x;
4    // Do stuff with v1
5    v = v1;
6    Array.getm(arr, idx, op, v);
7  } else {
8    int v2 = y;
9    // Do stuff with v2
10   v = v2;
11   Array.getm(arr, idx, op, v);
12 }
```

(b)

```
1  int v = 0;
2  if(...) {
3    v = x;
4    // Do stuff with v
5    Array.getm(arr, idx, op, v);
6  } else {
7    v = y;
8    // Do stuff with v
9    Array.getm(arr, idx, op, v);
10 }
```

(c)

Figure 5.17: Unifying Array Arguments: (a) shows a program that accesses an array with two different arguments that are not alive at the same time. (b) shows an inefficient way of unifying those arguments into a single variable `v`. (c) shows an optimized transformation in which the unified variable completely subsumes the original ones.

matched only once per control flow; if that match is at top level, this means they may only be matched once *per program*.

The second challenge is that a given dynamic table might be matched along only some of the control paths. This means we must have a way to sometimes *not* match each dynamic table. If the table were static, we could add an additional key to the table that disables it along specific control paths. However, dynamic tables are visible to the control plane, which expects them to have the same interface as in the original program. Adding an additional key changes that interface.

Instead, we allow each top-level `table_match` statement to be wrapped in a simple `if` statement describing its conditions for execution. When we compile the program to

P4, we evaluate these conditions using the special **gateway tables** on the hardware, which are able to evaluate certain simple conditionals without taking a stage. Hence the `if` wrapper does not increase the number of stages required by the program.

Unifying Lucid Table matches

We address both problems by unifying all matches for each table into a single match, which is placed at top level (but may be wrapped in an `if` statement). The transformation is somewhat complex, and is both described below and depicted in Figure 5.18. Line references in the description are to Figure 5.18b.

```
1  match ... with
2  | p1 -> {
3    x = do_something_p1();
4    int ret1 = table_match(tbl, x);
5    do_something_else_p1(ret1);
6  }
7  | p2 -> {
8    y = do_something_p2();
9    int ret2 = table_match(tbl, y);
10   do_something_else_p2(ret2);
11 }
12 | p3 -> { // No match here
13   do_something_p3();
14 }
```

(a)

```
1  int arg1 = 0;
2  int match_id = 0;
3  match ... with
4  | p1 -> {
5    x = do_something_p1();
6    match_id = 1;
7    arg1 = x;
8  }
9  | p2 -> {
10   y = do_something_p2();
11   match_id = 2;
12   arg1 = y;
13 }
14 | p3 -> { // No match here
15   do_something_p3();
16 }
17
18 int ret = 0;
19 if(match_id != 0) {
20   ret = table_match(tbl, arg1);
21 }
22
23 match match_id with
24 | 1 -> {
25   do_something_else_p1(ret);
26 }
27 | 2 -> {
28   do_something_else_p2(ret);
29 }
```

(b)

Figure 5.18: Unifying Table Accesses: (a) shows a program that matches against the table `tbl` in multiple places. (b) shows how to transform the code so that it only contains one match against `tbl`.

Assume we are targeting a particular table `tbl`. Without loss of generality, `tbl` is accessed inside a `match` statement `m`[8]. We create a new temporary variables for each argument to the `table_match`[9], as well as an integer variable `match_id`, which is initialized to 0 (lines 1-2). We then replace each access with assignments to the argument variables, as well as a new value of `match_id` for each access (lines 6-7 and 11-12). Lastly, we remove the remainder of the branch after each access, to be moved later in the program.

Next, we create a new `table_match` statement after the `match`, which accesses the table if `match_id` is nonzero, storing the result in a fresh variable(s) (lines 18-21). Finally, we create a new `match` statement that executes the remainder of the original branches, depending on the value of `match_id` (lines 23-29).

One might worry that such drastic reordering of the code might cause problems. Indeed, if `do_something_p3` involves accessing a global that appears after `tbl` in the pipeline, the resulting code will be rejected by our ordered type system! Fortunately, this will not cause compilation problems: unlike the type system, the compiler is not bound by the declaration order of globals and is free to reorder individual statements as it sees fit. Since the code from the `p3` branch is by definition independent of the code in the other two branches, the scheduler will always be able to order them appropriately (see §5.5 for more information).

5.4.4 Atomizing Operations

The next step of our compilation process is to ensure that each statement is atomic, as defined in §5.3.1. We do so in two steps: breaking up compound operations, and precomputing function arguments.

[8]We show how to transform `if` statements into `match` statements in §5.4.5. Due to the transformation in §5.4.1, we know that `tbl` is accessed inside at least one match statement.

[9]Note that every access to a given table must have the same number of arguments, which is the number of keys for that table.

Arithmetic Operations

Each arithmetic expression in the Lucid program will be computed on a Tofino ALU. These ALUs can only handle binary operations; accordingly, we must split up compound expressions like the one in Figure 5.19a. A naïve way to do this is shown in Figure 5.19b; however, this transformation is inefficient. Since each variable depends on the one defined before it, each addition must be evaluated in a different stage. As a result, it takes 3 stages to compile the code in Figure 5.19b.

```
1
2
3  int x = a + b + c + d;
```

(a)

```
1  int tmp1 = a + b;
2  int tmp2 = tmp1 + c;
3  int x = tmp2 + d;
```

(b)

```
1  int tmp1 = a + b;
2  int tmp2 = c + d;
3  int x = tmp1 + tmp2;
```

(c)

Figure 5.19: Breaking down arithmetic expressions: (a) shows a compound arithmetic expression. (b) shows one way to break it into successive binary operations. (c) shows a more efficient transformation for associative operations, which allows `tmp1` and `tmp2` to be computed in parallel.

Fortunately, for associative operations like addition, we have a better option. We can split the n-ary addition into a tree of binary additions, as shown in Figure 5.19c. Now we can compute `tmp1` and `tmp2` in parallel, so it only takes 2 stages to compute x instead of 3.

Note that we do not atomize boolean expressions in this way (although we do precompute them where necessary, as described below); boolean expressions will be eliminated entirely in the next pass.

Precomputing arguments

Functions (and hash expressions) require each argument to be immediate: either a constant or a variable. We enforce this by precomputing any non-immediate argu-

ments, as shown in Figure 5.20.

```
1
2
3  f(x, y + z, g(w))
```

(a)

```
1  int tmp1 = y + z;
2  int tmp2 = g(w);
3  f(x, tmp1, tmp2);
```

(b)

Figure 5.20: Precomputing Arguments: (a) shows a program that involves a function call with non-immediate arguments. (b) shows how we precompute the value of that argument so it can be replaced with an immediate variable.

5.4.5 Boolean operations

As discussed in 5.3.1, it is generally inefficient to evaluate boolean operations in ALUs like arithmetic expressions; instead, we should evaluate them using match-action tables. In Lucid terms, this means we must move all our boolean computation into the rules of a `match` statement[10]. However, `match` statements (and the match-action tables they represent) are only able to evaluate certain kinds of comparisons – specifically, equality tests between a single variable and a constant.

We ensure that all boolean computation happens in `match` statements using a series of three transformations:

1. Ensure that all boolean expressions occur in the condition of an `if` statement.
2. Convert each if-condition into disjunctive normal form (DNF), where the atoms have one of the following two forms:
   (a) `<var> == <literal>`
   (b) `<var> != <literal>`
3. Translate each `if` statement into an equivalent `match` statement.

[10]The exception are the `if` statements surrounding `table_match` statements (§5.4.3); converting these would result in nested match-action tables, which are disallowed by the hardware. These `if` statements are unaffected by the transformations in this section.

Converting boolean expressions to `if` statements

The first step is easy; the transformation is demonstrated in Figure 5.21. We simply place each boolean expression as the condition of an `if` statement, which then applies the appropriate result.

```
1 bool b = x and y;
2
3
4
```

(a)

```
1 bool b = false;
2 if (x and y) {
3   b = true;
4 }
```

(b)

Figure 5.21: Translating Booleans: (a) shows a program that assigns a boolean value. (b) shows how to translate that program so that the boolean computation occurs inside the condition of an `if` statement.

Normalizing `if`-conditions

The next step is to create the atoms of the DNF form by eliminating inequality operations and comparisons between two variables. We make use of the saturating subtraction operator `x |-| y` to do so, which returns `x - y` if `x >= y`, and `0` otherwise. Using a combination of saturating and regular subtraction, we can represent any (in)equality between two expressions as demonstrated in Figure 5.22.

```
1 bool gt = x >  y;
2 bool lt = x <  y;
3 bool ge = x >= y;
4 bool le = x <= y;
5 bool eq = x == y;
6 bool ne = x != y;
```

(a)

```
1 bool gt = x |-| y != 0;
2 bool lt = y |-| x != 0;
3 bool ge = y |-| x == 0;
4 bool le = x |-| y == 0;
5 bool eq = x  -  y == 0;
6 bool ne = x  -  y != 0;
```

(b)

Figure 5.22: Standardizing comparisons: (a) shows several (in)equality tests. (b) shows how each test can be replaced with a equality tests between an expression and a constant.

Since each atom must have a variable on the left, we precompute each subtraction operation, storing the result in a temporary variable. Then we translate each `if`-condition into disjunctive normal form.

From `if` to `match`

The final step is to convert each `if` statement into a `match` statement. Our previous step ensured that the condition of each `if` is in disjunctive normal form, and the atoms are all equality tests between a variable and a constant. We say that a disjunct is *negative* if any of its atoms is a non-equality test (using `!=`), and *positive* otherwise. If all disjuncts are positive, then the transformation is easy: each disjunct is a single rule, with a catch-all rule at the end, as shown in Figure 5.23.

```
1  if((x == 4 and y == 17)
2      or z == 42) {
3    do_something();
4  } else {
5    do_something_else();
6  }
```

(a)

```
1  match x, y, z with
2  | 4, 17, _ -> { do_something(); }
3  | _, _, 42 -> { do_something(); }
4  | _, _, _  -> { do_something_else(); }
5
6
```

(b)

Figure 5.23: Converting `if` statements: (a) shows an `if` statement with only positive atoms. (b) shows how that statement can be translated into a `match` statement.

If we have negative atoms, the transformation is trickier. Since `match` statements can only test equality, not non-equality, we must add additional rules to "screen out" bad values; Figure 5.24 shows how to do this if there is only one negative disjunct. The first two rules screen out any values where `z == 42` or `w == 97`; the remaining rules test the other atoms.

```
1  if(    x == 4
2     and y == 17
3     and z != 42
4     and w != 97) {
5    do_something();
6  } else {
7    do_something_else();
8  }
```

(a)

```
1  match x, y, z, w with
2  | _, _, 42, _ -> { do_something_else(); }
3  | _, _, _, 97 -> { do_something_else(); }
4  | 4, 17, _, _ -> { do_something(); }
5  | _, _, _,  _ -> { do_something_else(); }
```

(b)

Figure 5.24: Converting `if` statements: (a) shows an `if` statement with exactly one negative disjunct. (b) shows how to translate that `if` into a `match` statement by adding "screening" rules that check for specific values to avoid.

The most difficult part is combining multiple negative disjuncts. We cannot simply

concatenate all the rules into one large match, since matching an early "screening" rule for one disjunct might prevent us from ever checking the rules for the other disjunct!

```
if((x == 1 and y != 2) or
    (z == 3 and w != 4)) {
  do_something();
} else {
  do_something_else();
}
```

(a)

```
match x, y with // First disjunct
| _, 2 -> { do_something_else(); }
| 1, _ -> { do_something(); }
| _, _ -> { do_something_else(); }

match w, z with // Second disjunct
| _, 4 -> { do_something_else(); }
| 3, _ -> { do_something(); }
| _, _ -> { do_something_else(); }
```

(b)

```
// Incorrect translation!
match x, y, z, w with
| _, 2, _, _ -> { do_something_else(); }
| 1, _, _, _ -> { do_something(); }
| _, _, _, 4 -> { do_something_else(); }
| _, _, 3, _ -> { do_something(); }
| _, _, _, _ -> { do_something_else(); }
```

(c)

Figure 5.25: Translating `if` statements (incorrectly!). (a) shows an `if` statement with two negative disjuncts. (b) shows how to generate one `match` statement for each disjunct. (c) shows an incorrect way to combine the tables, by concatenating the non-default rules. This example fails if, for example, $(x, y, z, w) = (1, 2, 3, 0)$.

For example, consider the code in Figure 5.25. The disjuncts in Figure 5.25a will be translated individually into the `match` statements in Figure 5.25b. However, if we simply concatenate the non-default rules, we get the `match` statement in Figure 5.25c. This table does not have the same semantics as the original `if` statement; for example, if $(x, y, z, w) = (1, 2, 3, 0)$, the original statement will run `do_something()` but the new statement will immediately match the first rule, and run `do_something_else()`.

Merging matches To handle `if` statements with negative disjuncts, we begin by creating one `match` statement for each disjunct in isolation, using the strategy described above. We then apply a merging algorithm to iteratively combine these `match` statements into one. The algorithm computes a cross product of rules: for each rule

pair, it creates a new rule that executes the bodies of both input rules. If it finds certain rules are redundant or unmatchable, those rules are pruned.

```
if((x == 1 and y != 2) or
    (z == 3 and w != 4)) {
  do_something();
} else {
  do_something_else();
}
```

(a)

```
match x, y with // First disjunct
| _, 2 -> { do_something_else(); }
| 1, _ -> { do_something(); }
| _, _ -> { do_something_else(); }

match w, z with // Second disjunct
| _, 4 -> { do_something_else(); }
| 3, _ -> { do_something(); }
| _, _ -> { do_something_else(); }
```

(b)

```
match x, y, z, w with
| _, 2, _, 4 -> { do_something_else(); }
| _, 2, 3, _ -> { do_something(); }
| _, 2, _, _ -> { do_something_else(); }
// | 1, _, _, 4 -> { do_something(); }
// | 1, _, 3, _ -> { do_something(); }
| 1, _, _, _ -> { do_something(); }
| _, _, _, 4 -> { do_something_else(); }
| _, _, 3, _ -> { do_something(); }
| _, _, _, _ -> { do_something_else(); }
```

(c)

Figure 5.26: Translating `if` statements. (a) shows an `if` statement with two negative disjuncts. (b) shows how to generate one `match` statement for each disjunct. (c) shows the combined table, obtained by taking the cross product of the tables in (b). The commented branches are redundant and can be pruned.

An example of this process is shown in Figure 5.26. The two disjuncts in Figure 5.26a are translated separately into individual match tables (Figure 5.26b) using the techniques described above. Then, those matches are combined, taking a cross product of their rules. The body of each new rule is `do_something()` (the `then` branch) if either rule executed the `then` branch; otherwise, the body is `do_something_else()` (the `else` branch).

The algorithm we use is a specialized version of the general match-merging algorithm we will describe in §5.5.4. A key difference in our case is that while the general algorithm executes the bodies of both rules in each new rule, we only need execute one: either the `then` branch (`do_something` in our example), or the `else` branch.

Since there are only two possible rule bodies, we have more opportunities to prune redundant rules.

Complexity of merging In general, merging two tables is a multiplicative operation: combining tables with n and m rules will result in $n \cdot m$ rules, though we may be able to prune some of them as in Figure 5.26c. In our case, each initial table has one rule for each negative atom, plus one rule for *all* positive atoms, and finally a default branch. Thus if disjunct i has n_i negative atoms and we have k disjuncts, the final number of rules is

$$(n_1 + 2) \cdot (n_2 + 2) \cdots (n_k + 2) = O(n^k),$$

before pruning, where n is the maximum number of negative atoms in any disjunct.

Thus, in general, merging two tables is exponential in the number of disjuncts. However, in the specific case of `if` statements, we can do better: when merging in a disjunct with only positive atoms, we need not compute a cross product, but instead simply insert a single rule matching that disjunct at the top. This works because if the disjunct is true, we immediately know we should execute the `then` branch, without needing to consider the other disjuncts. Using this strategy, merging a positive disjunct into a table is actually a constant-space operation – it increases the size of the other table by 1, rather than doubling it (or more)!

This means we may exclude all positive disjuncts from our equation above; in other words, the number of rules generated is exponential in the number of *negative* disjuncts, not the overall number of disjuncts.

The complexity of this operation means that we are trading large `match` statements for the ability to compute a lot of conditions at once, saving stages. In general, this is a good deal; the Tofino has much more room for tables than it has stages. Still, it is useful to keep the number of negative atoms low when possible. Finding opportunities

to precompute them for "free" (i.e. in ways that do not increase the number of stages) is a promising direction for future work.

5.5 Scheduling Lucid

The final step of compilation is to assign each statement to a specific stage in the Tofino's packet processing pipeline. There are two primary constraints when doing so. First, we must respect dependencies between actions: if B depends on A then we must schedule B in a later stage than A. Second, we must respect the resources available in each stage of the pipeline, by not scheduling more in a stage than it can handle.

5.5.1 Immutable Conditions

```
1 match x with
2 | 10 -> {
3   y = y + 7;
4   z = 12;
5   match y with
6   | 11 -> {
7     w = 19;
8   }
9 }
```

(a)

```
1 match x with
2 | 10 ->
3     { y = y + 7; }
4 match x with
5 | 10 ->
6     { z = 12;    }
7 match x, y with
8 | 10, 11 ->
9     { w = 19;    }
```

(b)

Figure 5.27: (a) shows a simple program that conditionally executes some statements. (b) shows how we transform this program so that each statement is encased in a match statement that specifies the conditions for it to be executed.

During our scheduling algorithm, we will split up blocks of statements such that each statement is individually annotated with the conditions for its execution, as shown in Figure 5.27. A consequence of this transformation is that we re-check the condition at each stage, potentially resulting in code similar to that in Figure 5.28.

However, the transformation depicted in Figure 5.28b is incorrect! The first line of the `match` statement mutates `x`, which might cause us to execute line 2, but not

```
1 match x with
2 | 10 -> {
3   x = x + 1;
4   y = x + x;
5 }
```

(a)

```
1 // Unsound transformation!
2 match x with
3 | 10 -> { x = x + 1; }
4 match x with
5 | 10 -> { y = x + x; }
```

(b)

```
1 int x_orig = x;
2 match x_orig with
3 | 10 -> { x = x + 1; }
4 match x_orig with
5 | 10 -> { y = x + x; }
```

(c)

Figure 5.28: Breaking down `if` statements. (a) shows an `if` statement that modifies a variable in its condition. (b) shows an *incorrect* transformation of the `if` into individual annotated statements. The transformation is unsound because the condition `x < 10` may change between lines 2 and 3. (c) shows a sound transformation, which stores the original value of `x` in a fresh variable.

line 3, an impossibility in the original program. We solve this issue by storing the original value of `x` in a fresh variable, then testing that variable instead, as in Figure 5.28c. Since this variable is fresh, we are guaranteed that is is not mutated later.

To avoid creating unnecessary variables, we only perform this transformation if the condition might change while executing a branch of the `match` statement – that is, if one or more variables in the condition are mutated before the final statement in the branch.

5.5.2 Dependency graphs

The first thing we need to do when scheduling is to determine how the statements in the program relate to each other. We do so by constructing a series of graph representations of the program. An overview of the process for an example program appears in Figure 5.29. For clarity, we use `if` statements in the code, although in practice we would have transformed these into `match` statements before scheduling.

We begin by constructing a control-flow graph with two kinds of nodes: **action nodes**, which represent some computation, and **branch nodes**, which represent

branching control flow. Edges represent the ability for control to flow from one node to another, and are annotated with the conditions necessary for that to occur. This graph is represented in Figure 5.29(1); for brevity, statements are referred to by the names indicated in the comments of the code block. Note that since we have eliminated loops, and Lucid does not allow recursive functions, this is always a directed acyclic graph (DAG).

The next step is to annotate each node in the graph with its conditions for execution. We remove branch nodes by adding edges from their parent(s) to their successors, and annotate each successor with the conditions on the original edge, adding to any existing annotation. The result is a graph with only action notes, each of which is annotated with *all* the conditions to execute it, including prior control flow decisions. This graph is represented in Figure 5.29(2). If we converted this graph back to a Lucid program, it would have the form demonstrated in Figure 5.27b. Note that thanks to the transformation in Figure 5.28, we are guaranteed that this operation does not change the semantics of the program.

The final step is to turn the annotated control-flow graph into a dependency graph, in which an edge from statement `s1` to `s2` indicates that `s2` must be executed after `s1` due to a dependency. We consider three types of dependency: **read/write** (`s1` reads a variable that `s2` modifies), **write/read** (`s1` modifies a variable that `s2` reads), and **write/write** (both modify the same variable). We construct the graph by computing, for each node, the set of its descendants that directly depend on the variables it includes. We then add an edge to each element of that set. Transitive dependencies (i.e. `v1` depends on `v2`, which depends on `v3`) are implicit in the structure of the graph. The result of this step is depicted in Figure 5.29(3).

```
Array nexthops = new Array<<32>>(NUM_HOSTS);
Array pcts = new Array<<32>>(NUM_PORTS_X3);
Array hcts = new Array<<32>>(NUM_HOSTS);
memop plus(int cur, int x) { return cur + x; }

event count_pkt(int dst, int proto) {
    int idx = Array.get(nexthops, dst); // nexthops_get
    if (proto != TCP) {                 // if_0
        if (proto == UDP)               // if_1
            idx = idx + NUM_PORTS;      // idx_eq_0
        else
            idx = idx + NUM_PORTS_X2;   // idx_eq_1
    }
    Array.set(pcts, idx, plus, 1);      // pcts_fset
    if (proto == TCP)                   // if_2
        Array.set(hcts, dst, plus, 1);  // hcts_fset
}
```

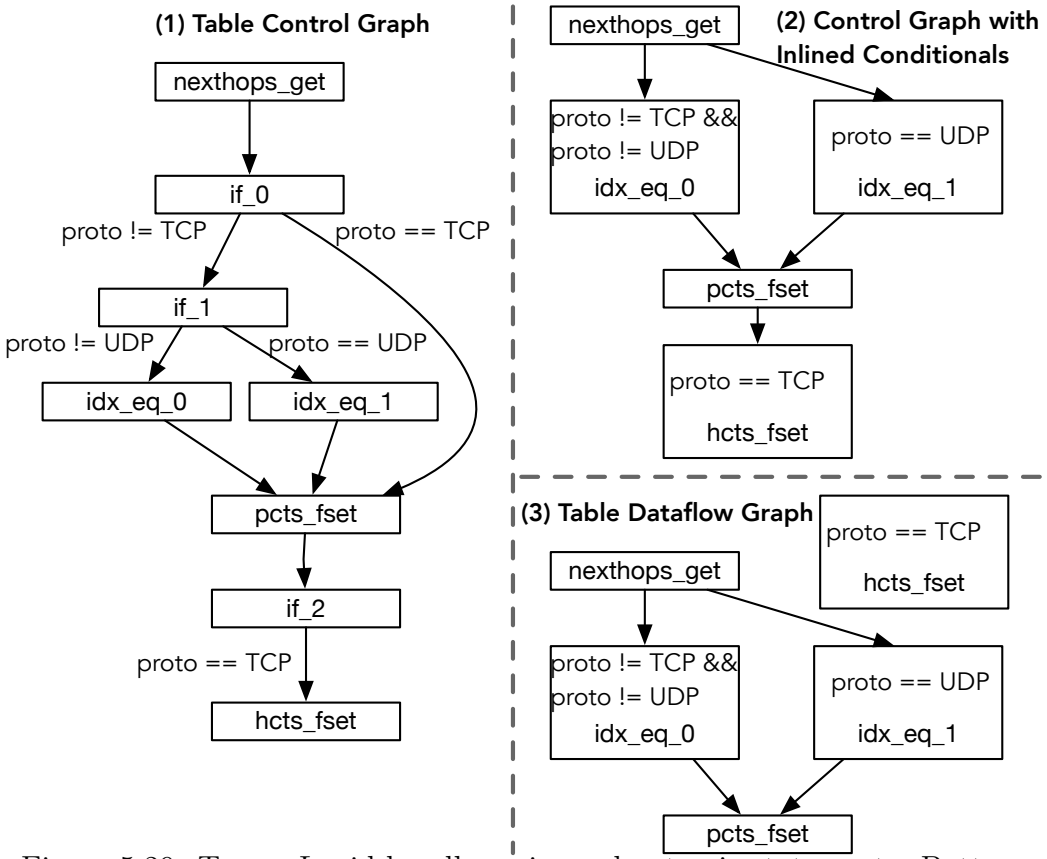

Figure 5.29: Top: a Lucid handler using only atomic statements. Bottom: (1) the handler represented as an control flow graph, (2) the same graph with execution conditions inlined, and (3) the graph optimized to require fewer pipeline stages. Image taken from original Lucid paper [73].

5.5.3 Layout Algorithm

The result of the previous process is a directed acyclic graph in which nodes are individual statements annotated with their conditions for execution, and edges represent direct dependencies between statements. Our final goal is to assign every node to a pipeline stage, such that each node is in a later stage than its predecessors, and such that all the nodes assigned to each stage fit in that stage's resource constraints (described below).

To do so, we treat each node as a very small match-action table, as depicted in Figure 5.28. We walk across the graph nodes in topological order, attempting to assign each node to the earliest stage its dependencies allow. If the stage has no room for more tables, we attempt to *merge* the node into an existing table in that stage, using the algorithm described in §5.5.4. If this fails for each existing table (because the merged table violates the resource constraints), we attempt to assign the node to the following stage instead, and continue in this way until we run out of stages or the node is successfully assigned.

Resource constraints We maintain an internal model of the resources available to a single table, as well as in each stage of the Tofino's pipeline. For tables, we track:

- The maximum number of statements in each of its actions,
- The maximum number of bits it can match (i.e. the sum of the sizes of the matched variables),
- The maximum number of different arrays it can access, and
- The maximum number of different hash units it can access.

For stages, we track:

- The maximum number of tables in the stage,

- The maximum amount of total memory that can be allocated to tables in this stage,
- The maximum number of different arrays that can be accessed from this stage,
- The maximum number of different hash units that can be accessed from this stage,
- The maximum number of bits that can be collectively emitted by hash units at this stage, and
- The maximum amount of total memory that can be allocated to arrays in this stage

There are some resource constraints that we do not model: in particular, we only consider *some* constraints regarding memory allocation within a stage. In addition to memory taken up by stages and arrays, there are some memory overheads for actions (in the form of pointers and parameters to those actions) and some overhead when memory in a stage is split up between multiple tables and/or arrays, which happens often. It is conceivable that our scheduler could model these constraints in the future, but doing so requires more work to understand and represent them. If the Tofino compiler finds that we have overfilled a stage, it will automatically shift tables down to later stages as needed.

Combining memory accesses Before we schedule our statements, we make one final tweak to the graph. Since each array must be stored in a single stage, all accesses to that array must happen in the same stage. We enforce this by combining all the graph nodes for each array access into one, merging the tables as described below. Here is the grand payoff of our ordered type system: because the arrays are accessed in order, it is impossible for there to be a dependency cycle between two different

arrays. Therefore, after merging these nodes, *the graph is still a DAG*, and so it is safe to walk over it topologically.

5.5.4 Merging tables

The key operation in our layout algorithm is merging two tables to create a single larger table. Doing so is essentially a cross-product operation: for each pair of rules, we create a new rule that combines the patterns and the bodies; if both patterns match, we execute both bodies. The basic transformation is illustrated in Figure 5.30; we use `match` statements to represent tables (both here and in our implementation).

```
1 match x with
2 | 1 -> { body_1(); }
3 | _ -> { body_2(); }
4
5 match y with
6 | 2 -> { body_3(); }
7 | _ -> { body_4(); }
```

(a)

```
1 match x, y with
2 | 1, 2 -> { body_1(); body_3(); }
3 | 1, _ -> { body_1(); body_4(); }
4 | _, 2 -> { body_2(); body_3(); }
5 | _, _ -> { body_2(); body_4(); }
6
7
```

(b)

Figure 5.30: Merging Tables: (a) shows a program with two tables that match on different variables. (b) shows how to merge those tables into one by taking a cross product of their rules.

Recall that per our layout algorithm, we will only try to merge two statements if their bodies are independent; hence we do not need to worry about the order in which we execute them in the bodies of the merged table. We also enforce the existence of a default branch for each `match` statement by adding one (whose body is a no-op) if necessary.

After merging two tables, we use a SAT solver (namely, Z3 [27]) to check to see if any rules are unreachable (because their conditions are subsumed by prior rules) or redundant (because they are immediately followed by rules with the same body and broader conditions). Examples of this are illustrated in Figure 5.31. In both cases, we prune any such rules that we find.

```
1 match x, y with
2 | _, 2 -> { body_1(); }
3 | 1, 2 -> { body_2(); } // Unreachable
4 | _, _ -> { body_3(); }
5
6 match x, y with
7 | 1, 7 -> { body_4(); } // Redundant
8 | 1, _ -> { body_4(); }
9 | _, _ -> { body_5(); }
```

Figure 5.31: Examples of rules that are unreachable (line 3) or redundant (line 7) and can therefore be pruned.

Combining expressions In Lucid, it is permissible to write an expression that matches a given variable multiple times per rule, e.g. `match x, y, x with ...`. Such expressions are disallowed in hardware match-action tables; each variable must appear in only one position of a given match. Hence if we are merging two tables that each match against the same variable, we must combine their patterns for that variable.

Doing so can be easily described piecewise. Recall that Lucid permits three types of pattern: wildcard patterns (`_`), which match anything, literal patterns (e.g. `0`) which only match that value, and bitstring patterns (e.g. `0b0**1`) which match integers bitwise, with `*` acting as a wildcard. For simplicity, we can treat literals at bitstrings with no wildcard bits, and treat wildcards as bitstrings with only wildcard bits.

To combine two bitstring patterns, compare them bitwise. For each bit:

1. If one is a wildcard bit, return the other one.
2. Otherwise, if the bits are the same, return that bit.
3. Otherwise, the patterns are contradictory, and cannot be combined. Hence the rule that required their combination cannot be matched, and so may be pruned.

An example of merging tables with overlapping expressions is given in Figure 5.32.

```
match x, y with
| 1, 2 -> { body_1(); }
| _, _ -> { body_2(); }

match y, z with
| 2, 3 -> { body_3(); }
| 4, 5 -> { body_4(); }
| _, _ -> { body_5(); }
```

(a)

```
match x, y, z with
| 1, 2, 3 -> { body_1(); body_3(); }
| _, 2, 3 -> { body_2(); body_3(); }
// Merge of 1, 2 and 4, 5 gets pruned
| _, 4, 5 -> { body_2(); body_4(); }
| 1, _, _ -> { body_1(); body_5(); }
| _, _, _ -> { body_2(); body_5(); }

```

(b)

Figure 5.32: Merging Tables: (a) shows a program with two tables that both match on the same variable `y`. (b) shows how to merge those tables by taking a cross product of their rules, and disregarding combinations with incompatible patterns.

5.6 Emitting P4

When we are done with the scheduling phase, we are left with a Core Lucid program that very much resembles a P4 program. The very last step is to actually create that P4 program, translating the Core Lucid constructs to their P4 counterparts. Because the scheduled program is so close to P4, this final step is little more than an assembler.

- **Arithmetic operations** are translated directly to P4, and executed on ALUs.
- **`match` statements** are translated to static match-action tables, whose rules are the patterns and whose actions are the associated code blocks.
- **Lucid Tables** are translated to dynamic match-action tables, and their interfaces are exposed to the control plane.
- **Actions** are translated to open functions (a P4 construct that behaves like a function, except they operate using the definitions of variables at their call site, not at their declaration site).
- **Hash statements** are implemented using the hash units provided at each stage.
- **Memops** are translated to RegisterActions.

- **Array updates** are implemented in stateful ALUs, using those RegisterActions.
- **Parsers** are translated to P4's own parser language; the body of each branch of each match statement is translated to a separate state.
- **Event arguments** are represented as special, Lucid-defined header fields of the packet.
- **Event generation** is implemented by setting the appropriate header values.
- **Event delays** are implemented using the Tofino's *pausable ingress queues*: delayed events are put in a special queue, and checked at regular intervals to see if their delay has expired.

5.7 Evaluation

We evaluate our compiler on two metrics:

1. **Speed**: are compilation times reasonable?
2. **Efficiency**: how many resources do compiled programs take?

To evaluate these, we compiled each of the sample programs from the Chapter 3 evaluation to a Tofino program. The results are summarized in Figure 5.33.

5.7.1 Compilation Time

As shown in Figure 5.33, all programs compiled fairly quickly, with most compiling in under a minute and even the longest compilation time below 5 minutes. Notably, the P4-to-Tofino compile times are shockingly low, with all programs finishing in under a minute. This stands in stark contrast to prior work, in which Tofino compilations

Application	Description	Time (sec) Lucid	Time (sec) P4	Tofino Stages
Stateful Firewall (SFW)	Blocks connections not initiated by trusted hosts. ***Control events update a Cuckoo hash table.***	208	54	10
Fast Rerouter (RR)	Forwards packets, identifies failures, and routes. ***Control events perform fault detection and routing.***	51	21	5
Closed-loop DNS Defense (DNS)	Detects/blocks DNS reflection attack with sketches & Bloom filters. ***Control events age data structures.***	30	26	8
*Flow [74]	Batches packet tuples by flow to accelerate analytics. ***Control events allocate memory.***	30	26	12
Consistent Shared State (SRO)[86]	Strongly consistent distributed arrays. ***Control events synchronize writes.***	17	<1	4
Distributed Prob. Firewall (DFW)	Distributed Bloom filter firewall. ***Control events sync. updates.***	14	23	7
+Aging	***Adds control events for aging.***	61	30	9
Single-dest. RIP	Routing with the classic Route Information Protocol (RIP). ***Control events distribute routes.***	17	14	6
Simple NAT	Basic network address translation. ***Control events buffer packets and install entries.***	10	12	5
Historical Prob. Queries (CM)	Measures flows with sketches for historical queries. ***Control events age and export state periodically.***	12	15	3

Figure 5.33: Compilation information for the applications from the original Lucid paper, originally shown in Figure 3.16. The first "Time" column indicates the time for our compiler to compile the Lucid program (to P4). The second column indicates the time for the Tofino compiler to compile that P4 program to a binary. The final column details the number of pipeline stages used by that binary.

can take several minutes or hours – in one extreme case, the compiler took multiple *days* to produce a working binary [37]. In that light, even the 5-minute compile time seems quite reasonable! In addition, the low overall compile times provide evidence that the Lucid compiler's optimizations really work, and provide the Tofino compiler with a much easier compilation problem.

5.7.2 Resource Efficiency

When compiling a dataplane program, there is one overarching goal: *make it fit*. Unlike conventional programs, "larger" dataplane programs (in terms of number of stages, size of tables, etc.) do not take longer to execute; it takes a packet the same amount of time to pass through a pipeline regardless of how much work that pipeline is doing. As a result, the primary metric for success of a compiler is its ability to fit a program within the pipeline, not necessarily its ability to make the program "small".

Nonetheless, the size of the resulting program is a good benchmark of the compiler's optimality: while the difference between 5 and 8 stages may be unimportant, the difference between 10 and 11 stages is enormous if the hardware only provides 10 stages. Furthermore, all compiler resources are "contained" by pipeline stages (e.g. the amount of memory is measured *per stage*, not as a total amount). Hence the number of stages used by a program is an reasonable measurement of that program's overall resource efficiency.

Figure 5.33 reports the number of stages used by each of our programs. All of them fit within the Tofino's pipeline. In practice, we have found that the Lucid compiler is not optimal: that is, it sometimes takes more stages than a human would for the same program. We have found several common sources of inefficiency; some can be dealt with automatically via improvements to the compiler, some are difficult to solve automatically but can often be easily fixed by users, and some are idiosyncratic and may not be an instance of a more general class of problem. A non-exhaustive list of common problems, along with potential fixes, appears below.

PHV Allocation Packet header vectors (PHVs) are the header locations in a packet in which local variables are stored; in several ways, they are analogous to registers in traditional computers. There are several constraints on how PHVs are used, which are very complex and not represented in Lucid (and so are not accounted for during

compilation). Issues arise when too many interdependent variables are alive at the same time, and can often be solved by the user adding hash statements, which are able to create fresh copies of variables without the dependencies of the original; an example is shown in Figure 5.34.

Performing this transformation automatically is difficult, since we have a limited number of hash units, and determining where to use them is nontrivial. A different tactic to minimize PHV errors is to aggressively try to overlay different variables that are not alive at the same time, in order to reduce the total number of PHVs used. This could be particularly useful when unifying arguments to an array update or `table_match`, since overlaying the arguments allows us to entirely avoid declaring a temporary variable.

```
1 // PHV Error: x1, ..., x20 are
2 // all alive at the same time,
3 // and interdependent
4 int tmp1 = x1 + x2;
5 int tmp2 = tmp1 + x3;
6 int tmp3 = tmp2 + x4;
7 ...
8 int tmp19 = tmp18 + x20;
```

(a)

```
1 int tmp1 = x1 + x2;
2 int tmp2 = tmp1 + x3;
3 int tmp3 = tmp2 + x4;
4 ...
5 // Identity hash
6 int tmp10' = hash(1, tmp10);
7 ...
8 int tmp19 = tmp18 + x20;
```

(b)

Figure 5.34: Fixing PHV Errors: (a) shows a program in which 20 different variables need to be placed in the same PHV container. (b) shows how to break this dependency chain using an identity hash. For the purposes of PHV constraints, `tmp10'` does not depend on any of the prior `tmp` variables (although of course it retains a semantic dependency on those variables).

Inlining hash statements In the hardware, hash units are situated earlier in each stage than tables and arrays. This means that sometimes, if a hashed value is used as a key to an array, it is possible to compute the key and then access the array in the same stage. Lucid does not currently perform this optimization, but doing so would not be theoretically challenging.

Maximizing memops Often, a user will want to retrieve a value from an array, compare it to another value, and branch on the result (as in Figure 5.35a). Doing so naïvely takes three stages: one to get the value, one to precompute the comparison, and one to branch on the result. This could be shortened to two stages by writing a memop that includes the comparison as part of the array update, as shown in Figure 5.35b. One could imagine performing this transformation automatically, though it would require an analysis to determine where it is possible.

```
1  int x = Array.get(arr, idx);
2  // Note: must be precomputed
3  if (x < y) {
4      ...
5  }
```

(a)

```
1  // Memops always return an int
2  memop less(int a, int b) {
3    if (a < b) {
4      return 1;
5    } else {
6      return 0;
7    }
8
9  int b = Array.getm(arr, idx,
10                     less, y);
11 // No precomputation necessary
12 if (b == 1) {
13   ...
14 }
```

(b)

Figure 5.35: Memop Optimization: (a) shows a program that retrieves a value from memory, then compares it to another variable. (b) shows how the program could be optimized (saving a stage) by performing the comparison simultaneously with the retrieval.

Uninitialized variables To prevent undefined behavior, Lucid does not permit users to declare uninitialized variables. In P4, however, variables are declared and initialized separately, and the initialization may take an extra stage. For example, the code in Figure 5.36a will assign `x` to `0` in one stage, and then immediately overwrite it in the next stage if `cond` is true. We could instead perform both operations in one stage by permitting undefined variables in Core Lucid, and ensuring that the variable is assigned along every control flow path, as shown in Figure 5.36b.

```
int x = 0;
if (cond) {
  x = 10;
}
```

(a)

```
int x;
if (cond) {
  x = 10;
} else {
  x = 0;
}
```

(b)

Figure 5.36: Avoiding Initialization Overhead: (a) shows a program that initializes the variable `x`, then immediately reassigns it. (b) shows an optimized program that does not spend an extra stage assigning the initial value to `x`.

Better heuristics Our current scheduling algorithm is greedy, and uses a heuristic to determine which tables to attempt to merge. It is possible that we could improve its performance with a better heuristic.

Precomputing negative atoms As discussed in §5.4.5, the number of rules we produce when converting an `if` statement to a `match` is exponential in the number of negative atoms in the condition. Although tables with many rules are no slower to execute, it takes significantly longer to merge them with other tables, and they take up more room in the stage. We could try to reduce the size of our tables by precomputing negative atoms, as shown in Figure 5.37. Doing so would allow us to attempt more merges in the same amount of time during compilation, potentially allowing us to discover more efficient configurations.

```

if (... and x != 10 and ...) {
  ...
}
```

(a)

```
bool tmp = (x != 10);
if (... and tmp == true and ...)
    {
  ...
}
```

(b)

Figure 5.37: Precomputing Atoms: (a) shows a program that involves an `if` statement with a negative atom. (b) shows how that atom could be turned into a positive atom using precomputation.

Doing this naïvely incurs a cost, unfortunately; the precomputation takes up a stage, which is a high price to pay. We may be able to avoid this cost sometimes, however. It may be the case that there is a "free" stage in which the variable is unused

before the comparison; in this case, adding a precomputation in that stage would not increase the total number of stages. Alternatively, the Tofino provides a special kind of **gateway table**, which can compute certain types of comparison without taking a stage. Determining which, if any, of these options applies to a given negative atom is not trivial, but is certainly possible with a sufficiently sophisticated analysis.

5.7.3 Comparison to the Tofino Compiler

A unique feature of the Lucid compiler is it not only generates P4 code, but it *specializes* that code for the Tofino. To do so, it takes on some of the compilation burden that is normally handled by the Tofino compiler. In particular, by transforming Lucid programs into a series of `match` statements, then scheduling and merging them, the Lucid compiler is providing the Tofino compiler with a much easier allocation problem.

Indeed, prior versions of the Lucid compiler did not have these transformations, and were much less successful. The first iteration did not translate `if` statements to `match`es, instead using ALUs to compute the `if` conditions. However, we found that the hardware constraints of `if`s on the Tofino were extreme and caused programs to take far too many stages.

The next iteration translated `if` statements to `match` statements, but did not merge or schedule them. In this case, we found that the Tofino compiler had immense difficulty compiling even moderate-complexity Lucid programs, such as the stateful firewall. As a rule of thumb, we found that layout would typically begin to take very long amounts of time and/or fail outright once programs had around 10 arrays and 4-5 P4 `if` statements. It was only once we began scheduling the tables ourselves, and merging them when possible, that we were able to compile more sophisticated Lucid programs.

Tofino Layout Algorithm It is clear that the Lucid scheduling algorithm works on more complex programs than the one the Tofino uses. Unfortunately, the Tofino compiler is proprietary, so we cannot directly compare our algorithm to theirs. However, we can compare to the algorithms laid out in Jose et al. [42], which evaluates two types of algorithm for compiling programs to programmable switch hardware like the Tofino: greedy, and Integer Linear Programming (ILP). In general, they found that ILP-based algorithms produced more optimal results (e.g. fewer stages), but took significantly longer than greedy algorithms like ours.

Given our observed behavior that the Tofino compiler begins to slow down rapidly as programs get larger, it is plausible that it uses some sort of exponential search algorithm like an ILP solver. It also considers more types of resource than the Lucid compiler, contributing to the complexity of the problem. However, it is unlikely that the compiler is merging tables the way Lucid does; in fact, we know of no other dataplane language that does so, although there has been some work on P4 compilers that utilize table merging or similar program transformations [49, 33]. It seems likely that the reason for our greater success is our ability to combine tables, as well as potentially our ability to schedule the program using Lucid's higher level of abstraction.

5.8 Related Work

Synthesis techniques Other dataplane programming languages ship with their own compilers, which use various techniques to fit their programs into the hardware. The P4 compiler [14] uses a "standard" compiler design in which the program is iteratively transformed by a series of passes, until it is eventually passed to a target-specific backend. The Domino compiler [70] breaks programs down into atomic *transactions* that can be implemented in a single hardware operation. The Lyra compiler [31]

takes in a description of the network and automatically splits the program across multiple devices. The P4All [36] compiler uses an Integer Linear Programming solver to automatically allocate memory to data structures.

A common thread among existing compilers (with the notable exception of the P4 compiler) is the use of of synthesis techniques to generate working or even optimal code. P4All uses its ILP solver; Domino and Lyra both use SAT solvers to synthesize their final layouts. However, their synthesis techniques are *all-or-nothing* – either they find a working configuration, or compilation fails wholesale. This makes it difficult to debug failed programs. In contrast, Lucid's approach is closer to the one used by the P4 compiler – a series of transformation passes culminating in target-specific code.

Jose et al. [42] examined the effectiveness of various generic compilation techniques to reconfigurable pipelines like the Tofino. They found that greedy approaches like Lucid's were generally faster than synthesis-based ILP approaches, but that synthesis unsurprisingly produced better final solutions for compiling programs.

VLIW The Tofino is an instance of a Very Long Instruction Word (VLIW) architecture [29]: each stage has multiple computation units (ALUs, sALUs, hash units, etc), and each operation must be allocated to one of them. In particular, the Tofino is an instance of a *clustered* VLIW architecture, in which registers (or PHVs, in the case of Lucid) are associated with specific computation units. Compilation to clustered VLIW architectures has been well-studied in the realm of traditional computing [9, 25, 66, 88]. Unfortunately, there are some critical distinctions between the traditional setting and the Tofino; in particular, most existing work assumes that it is acceptable (though undesirable) to spill registers to memory. In Lucid, if we run out of PHV space, the program is instead uncompilable. This necessitates different approaches to try to ensure programs compile.

Table Merging The merging of tables is one of the most crucial parts of Lucid's algorithm, but it is done via a simple greedy algorithm. Such compression strategies have been studied before. Smolka et al. [71] describes a compiler for NetKAT that represents routing tables using *Forwarding Decision Diagrams*, a generalization of Binary Decision Diagrams [16] specialized for networks. Use of this representation can potentially reduce the amount of space required to represent a given ruleset. Furthermore, Pous [61] shows how to use BDDs to check equivalence of expressions.

Recent work on Cetus [49] and Cat [33] has resulted in P4 compilers which utilize table-merging strategies similar to Lucid's. Cetus sometimes merges tables from different stages into a single stage, using the same strategy as Lucid. Cat uses table merging to combine nested if-else statements into a single flat table, analogous to Lucid's boolean elimination passes. A key difference between Lucid and related work is that Lucid programs typically start with many extremely small tables (most containing one statement each), while P4 programs start with fewer, larger tables written by the user. This means that the Lucid compiler *must* combine tables more aggressively than P4 compilers (since the total number of tables is limited), but also that Lucid has more flexibility in how it merges them.

A related topic to table merging is the problem of compressing rules in routing tables, or data structures more generally. This has been a topic of interest as the internet grows ever larger [43], and has led to several papers describing possible compression strategies [63, 64, 44]. Many of these techniques are not directly applicable, as tables in Lucid programs can represent many more things than routing information, but are nonetheless of interest as a way of possibly compressing the size of the tables.

Chapter 6

Parasol: Optimizing Dataplane Programs in Lucid

6.1 Dataplane Optimization

In comparison to P4, Lucid makes it much easier to write working dataplane applications and compile them to switches. However, tuning these applications for real-world use is still a challenging task. There are numerous decisions that must be made while writing a program – some relatively small, like a timeout for firewall entries, and some large, like which data structure to use to store traffic measurements. Different choices can result in wildly different performance[1], and the first program written is rarely the best one.

As a result, deploying dataplane programs can be a time-consuming and tedious task, which typically involves iteratively compiling, deploying, and tweaking the program until it both fits within the hardware and has acceptable performance. This process can take days or even weeks, and is made all the more frustrating since each step of the process can individually be time-consuming. A state-of-the-art P4 com-

[1] As measured by how well the program achieves its objectives, e.g. the accuracy of measurement program, or the hit rate of a cache.

piler might take minutes, hours, or even *multiple days* to produce a result [37]. Even prewritten programs are not immune, since they may need to be altered to fit on hardware with different restrictions, or tuned to work in a network with a different topology or traffic pattern.

To alleviate these problems, researchers have developed a variety of different program synthesis systems aimed at programmable switches [70, 32, 36, 31, 83, 91]. These systems either manipulate existing programs to fit them onto switches, or lift the level of abstraction at which the program is written, giving it enough "flex" to be automatically fit into switches by optimizing the use of key resources such as memory, ALUs, or pipeline stages.

As useful as they are, these systems only scratch the surface of what is possible. Each of the above systems is limited in one or more of the following ways.

Limited objectives. Most systems to date focus on optimizing simple on-switch resources, such as memory footprint, number of pipeline stages, or ALU usage. However, operators typically evaluate applications on far more sophisticated criteria than just whether they happen to fit into the switch pipeline. Accuracy of measurements, effectiveness relative to an idealized model, and bandwidth used are just a few other ways to evaluate data-plane algorithms. Indeed, even the most memory-intensive applications are typically developed to optimize some *other* criteria. For instance, NetCache [41] – an in-network cache for key-value stores – uses several data structures, including a count-min sketch (to identify popular keys) and a multi-stage hash table (to cache the values for popular keys). While optimizing memory layout of these data structures is important, the high-level objective is actually to maximize cache *hit rate*. No tool to date has the ability to specify objectives at such a high level of abstraction.

Limited "program flex." To give optimizers a chance to improve a program implementation, they must be free to change that implementation – the more freedom (i.e., the more "flex" in the program), the more opportunities an optimizer has to make improvements. Standard optimizing compilers have very little freedom in this regard; they must preserve the surface-level semantics of programs. A system like P^{2}GO [83] deviates from this requirement by cutting out program components that are unnecessary for processing a particular traffic trace, but this carries some risk if traffic not present in the trace shows up in the live network. Even if it does not, P^{2}GO has limited ability to make changes, because it utilizes only three operations: it can merge tables, remove dependencies, or move processing to the control plane.

A system like P4All [36] or SketchGuide [91] adds a little more flexibility by allowing data structures to be resized. Still, memory allocation is not the only thing that affects performance: other crucial parameters include the rate at which active probes are emitted in a telemetry application, the choice of data structures to use in an in-network cache, the threshold at which to declare a heavy hitter, or the criterion to use for failure detection. No tool to date allows users to write programs with so much flex, let alone automatically optimize them.

Limited environmental input. The performance of a data-plane application is a product of its environment. A program tailored to one workload may not perform well if used in a different setting. As a result, systems like Chipmunk [32] and P4All [36] are limited because they have no access to traffic traces. Even if it were possible to express a property such as "optimize cache hit rate" in a system like P4All (which it is not), it would not be possible to completely solve the optimization problem because hit rate depends on the distribution of requests in the network, which P4All does not consider. P^{2}GO and SketchGuide *do* provide access to such data, but they have neither the flex nor the range of objectives to exploit that information to its fullest

potential.

The fundamental difficulty of optimization is the tradeoff between expressibility and complexity. The more flex a program has, the more difficult it is to optimize, as the program can have any number of parameters that affect its performance, and it can be nearly impossible to develop objective functions to capture every parameter in a program. On the other hand, programs with little or no flex leave little or no room for optimization in the first place, rendering an optimizer useless.

6.1.1 Parasol

In this chapter, we will discuss Parasol, a novel, general framework for data-plane-program optimization that builds upon Lucid to overcome the limitations of earlier frameworks. Parasol programmers write *sketches* [72] in Lucid, which are normal programs with several "holes". These holes represent the *parameters* of the program; each is an undefined value that will be filled in by the optimizer. Parameters in Parasol are highly flexible – they can control just about any aspect of the implementation. This might include memory layout, decision thresholds, measurement intervals, or even a choice between data structures; in contrast, systems like P4All and SketchGuide are limited to only optimizing memory layout.

The sketch is then passed to an optimizer, which uses an iterative search algorithm to automatically optimize the parameters according to a user-defined objective. The program sketch is simulated in the Lucid interpreter with a candidate set of parameter values, during which time the user's objective function may take arbitrary measurements of the network. Upon completion, the objective function computes a score for those parameter settings, and the algorithm uses the past scores to choose a new set of candidate values. This process repeats for a specified number of iterations, after which the optimizer returns the highest-scoring parameter values that

successfully compile to the hardware.

Both parts of the objective function – measurements and score computation – are written in Python, rather than the more limited languages of switch data planes. Hence, users can express essentially unlimited optimization criteria – the main constraint is the fidelity of the simulation environment to reality. Furthermore, the optimizer simulates the program's behavior on a specific traffic trace, allowing the optimizer to tailor the performance to a particular networking environment.

To summarize, Parasol is a new dataplane optimization framework with the following features.

- **Flexible objectives:** Parasol's optimization algorithm can optimize for a wide variety of *high-level* metrics such as hit rate or measurement accuracy.
- **Flexible programs:** The parameters of a Parasol program may control many properties, including probe generation frequency, algorithmic choices, memory layout, data-structure selection, or threshold values.
- **Flexible environments:** Parasol programmers may tailor their optimization to particular network environments by providing representative traffic traces.

We have evaluated Parasol by developing a number of data-plane programs with various parameters and objective functions. Our experiments found that the Parasol optimizer completed an iteration in approximately eight minutes on average (with an average trace size of two million packets), and all applications could be optimized with a time budget of two hours. The solutions produced by the optimizer not only complied with the resource constraints of the hardware, but were comparable in performance to hand-optimized P4 code.

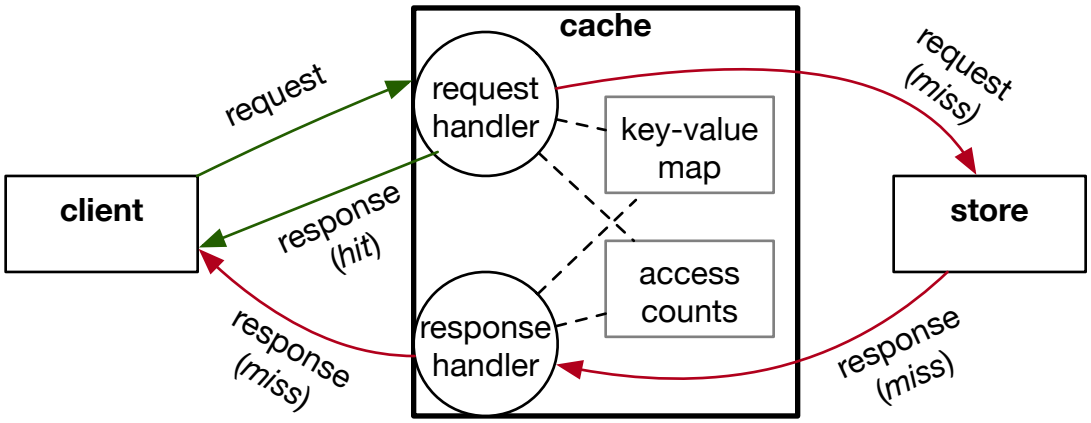

Figure 6.1: Motivating example: an in-network cache.

Param	Description
C_m	Number of columns / hashes in multi-hash table (MHT).
R_m	Number of rows (cells per hash) in multi-hash table.
C_c	Number of columns / hashes in count-min sketch (CMS).
R_c	Number of rows in CMS.
T_t	Timeout threshold for cache.
T_r	Replacement threshold.
P	Use precision in place of MHT + CMS.

Figure 6.2: Parameters of the data-plane cache.

6.1.2 Attribution

The work on Parasol was spearheaded by Mary Hogan as an extension of her prior work on dataplane optimization [36]. Other contributors were the author, David Walker, John Sonchack, Jennifer Rexford, and Shir Landau-Feibish. The author's contributions were in designing and implementing the extensions to the Lucid language and interpreter described in this chapter, as well as providing input to the overall design of Parasol. The text of this chapter is adapted with minor modifications from a draft version of the Parasol paper [37].

6.2 An Illustrative Example

As in earlier chapters, before we describe Parasol in detail, we will provide a running example to highlight its various components. Our example is a load-balancing cache, inspired by NetCache [41], which a database operator might wish to deploy in their network. The structure of the cache is illustrated in Figure 6.1. The cache reduces load on database servers by directly responding to requests for the most popular keys, and forwarding only cache misses to the servers.

The cache operates by storing key/value pairs in a hash table on a switch. When a request arrives, the switch first checks to see if the key is in the table; if it is, the switch simply retrieves the value and sends it back to the requester. Otherwise, the switch forwards the request to the appropriate storage server. When the response arrives, the switch forwards it to the client and optionally caches the entry.

To maximize efficiency, the cache should store values for the most popular keys. Because popularity may change over time, the switch dynamically updates its cache to remove less popular keys in favor of more popular ones. To enable this, the switch tracks statistics about the popularity of keys *not* stored in the cache using a second data structure: a compact, approximate counter (e.g., a count-min sketch (CMS)).

Parameters and performance. The high-level description of the cache algorithm is quite simple, but to implement it, a programmer must make numerous low-level decisions. How much memory should be allocated to the hash table vs. the counter? When should we replace cached keys? How should we represent the counter – using a CMS, or something like Precision[2] [10]? Each of these questions corresponds to a *parameter* of the program; Figure 6.2 provides a non-exhaustive list of the parameters that an implementation might depend on.

These decisions are not simply details – they can have significant performance

[2]A hash table that probabilistically replaces cached keys upon collision, where more popular items are less likely to be replaced.

implications. For example, a larger hash table can cache more keys at once, but reduces the memory available for the approximate counter and, in turn, its accuracy. A too-small timeout means that moderately popular keys will get frequently evicted and re-added, while a too-large timeout can result in less popular keys staying in the cache for far too long.

Unfortunately, predicting the behavioral effects of the parameters is difficult, because different parts of the program are competing for the extremely limited resources provided by the switch. We can predict that allocating less memory to the approximate counter will reduce its accuracy, but perhaps the extra space for the cache itself will outweigh that negative – or perhaps not. Trying to derive theoretical high-level behavioral implications from low-level parameter values is an extremely difficult task. Even then, the theoretical behavior will have to make assumptions about the environment in which the program is run [26]. *Even if* the programmer goes through the considerable effort of working out a closed-form objective function for a cache, it typically only expresses worst-case or average-case performance; the actual rate may be drastically different in practice [22].

In contrast, the *desired* behavior of a data-plane cache is easy to define – it should maximize hit rate. This behavior is equally easy to measure, by simply monitoring the switch in question and recording whether each incoming packet is a hit or a miss. While it would be very difficult (likely impossible) to derive a closed-form equation that relates the cache's hit rate to its parameters (a necessary step for using ILP-based optimization frameworks [36]), a simulation-based approach lets us simply observe the cache's behavior in practice.

Traffic dependence. There is another wrinkle: the hit rate of the cache does not depend solely on the parameters, but also on the network. Specifically, the hit rate depends on which keys are in the cache, which is determined not only by the size of

the data structures but also the choice of data structure in the first place (e.g. CMS vs. Precision), the timeout parameter, and the traffic flowing through the network.

P4All [36], another framework that optimizes parameters of a program sketch, is only able to model the first of these parameters (the size of the data structures). As a result, the performance of the P4All program is limited by the programmer's hard-coded choices for the remaining parameters. Unfortunately, certain parameters can have a large range of potential values (e.g., timeout could range from milliseconds to seconds to even longer), and the subset of that range that performs well in practice may be quite small, making it easy to pick suboptimal values.

Indeed, we found during testing that if the programmer chose values poorly, the hit rate for a skewed workload could be as low as 56%, while our optimizer produced a solution with a hit rate of 93%. For a uniform workload the minimum hit rate plunged to 11%, while Parasol managed a hit rate of 28%[3]. Clearly, these additional parameters *do* have substantial performance impacts in practice, meaning that the ability to optimize them is important.

6.3 Extensions to Lucid

Parasol employs an extended version of Lucid as a *sketching language* that allows users to write parameterized programs. We chose Lucid as the basis for Parasol for two reasons. First, as a high-level language, it provides useful abstractions for representing the numerous decisions programmers must make during implementation. Second, Lucid's two backends (the interpreter and P4 compiler) are ideal for Parasol's use case. The interpreter can simulate a program's behavior without a lengthy compilation process, while the compiler can ensure that parameter values are practical (i.e. they actually compile to the hardware)

[3]One might worry that Parasol is achieving its better hit rates by overfitting to its input trace; this is a concern for any framework that relies on a particular input. We discuss how to prevent overfitting in §6.4.3.

To implement Parasol, we add three new features to Lucid. First, we add symbolic values (à la P4All [36]) to represent the parameters of a program that should be optimized. Second, we add a way to select between two different data structures based on a symbolic value. Finally, we add a foreign function interface that allows the user to take arbitrary measurements of the network during simulation. Figure 6.3 shows a pared-down example implementation of a dataplane cache that we use to demonstrate these extensions. Parts of the program that do not relate to Parasol's extensions have been omitted, including the hash table storing the key/value pairs.

Symbolic Values. Symbolic values in Parasol function as placeholders that may take on any value of the given type. Each is later replaced with a concrete value, supplied during the compilation/optimization process. Once declared, a symbolic is used in the same way as a compile-time constant.

The program in Figure 6.3 contains four symbolic values. The boolean `useCms` determines if the program should use a CMS or Precision data structure, and the integer `trackerSize` determines how much memory is allocated to that structure. If a CMS is used, `cmsThresh` determines the threshold for adding new keys to the cache. Finally, `timeout` determines when keys in the cache are considered expired.

Selecting Data Structures. In Figure 6.3, the CMS and Precision data structures are each represented as a module containing a type definition, constructors, and functions for deciding when to add a particular key to the cache. However, these modules are not directly referenced in the rest of the program. Instead, the program refers to the `KeyTracker` module, which is an alias for either `CMS` or `Precision`, depending on the symbolic value `useCms`. When the program calls the function `KeyTracker.create` to initialize the tracker, the appropriate constructor is called, and similarly for the function `KeyTracker.DecideIfAddingKey`.

Parasol's extension to the Lucid type checker makes sure `CMS` and `Precision`

```
symbolic bool useCms;
symbolic int trackerSize;
symbolic int cmsThresh;
symbolic int timeout;

module CMS : {
  type t = ...;
  fun t create(int size) {...}
  fun int getCount(int key) {...}
  fun bool decideIfAdding(int key)
    { return (getCount(key) > cmsThresh); }
}

module Precision : {
  type t = ...;
  fun t create(int size) = {...}
  fun int getCount(int key) {...}
  fun bool decideIfAdding(int key) {...}
}

module KeyTracker = CMS if useCms else Precision;

global KeyTracker.t tracker = KeyTracker.create(memSize);

extern logHits(bool found);

event request(int key) {
  int cachedKey =   // Hash key to produce an index,
  int cachedTime =  // and return the value at that index
  int cachedValue = // in the hashtable

  bool found = (key == cachedKey);
  int timeDiff = Sys.time() - cachedTime;
  bool expired = timeDiff > timeout;

  logHits(found);
  if (found)
    { generate response(cachedValue); }
  else if (expired)
    { AddKeyToCache(key); }
  else {
    bool add = KeyTracker.decideIfAdding(key);
    if (add) { AddKeyToCache(key); }
  }
}
```

Figure 6.3: A demonstrative implementation of a data-plane cache in Parasol. Parts of the code not containing novel elements have been truncated or omitted entirely.

implement the same interface, which allows the program to use `KeyTracker` safely while remaining oblivious to the implementation-level differences between the two. If the modules differed, the programmer could create wrapper modules to ensure they present the same interface.

Foreign Function Interface. Our final extension lets a programmer instrument their code with calls to external measurement functions that are executed by the Parasol simulator, but removed from the final compiled program. In Figure 6.3, the extern `logHits` is a function implemented in Python by the programmer, which counts the number of cache hits and misses while the Parasol simulator is running. Each time a cache lookup is performed, `logHits` is called to record whether the lookup was a hit or a miss. After completing a simulation, the Parasol optimizer can compute the hit rate simply by dividing the number of hits by the total number of lookups.

Parasol does not impose any restrictions on what can be passed as a parameter to extern functions, but permits only functions that have no return value. Since externs cannot modify any Lucid program state, this means they can be safely elided during compilation.

6.4 Optimizing Sketches

Once a sketch is written in Lucid, the next step is to optimize its parameter values. A high-level overview of Parasol's optimization framework is provided in Figure 6.4. The programmer provides four inputs: (1) a program sketch, (2) a traffic trace, (3) one or more *measurement functions*, and (4) an *objective function*. The Parasol optimizer then finds effective values for the parameters of the sketch using an iterative search algorithm. In each iteration, the search algorithm selects a concrete value for each symbolic value. The resulting program is then simulated on the provided traffic trace

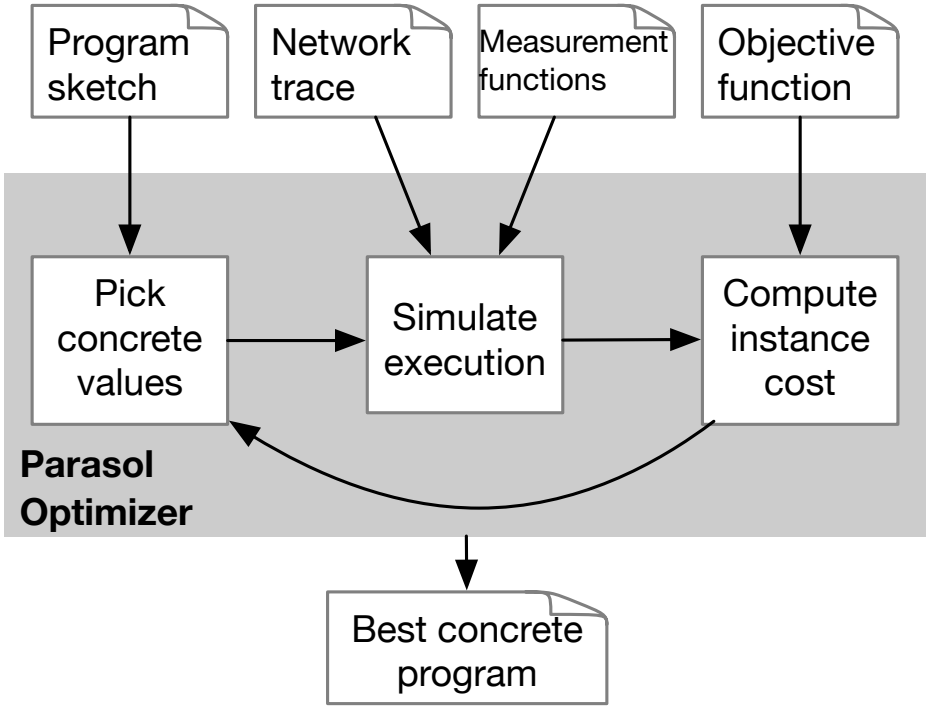

Figure 6.4: Overview of the Parasol optimization framework.

using the Lucid interpreter.

During simulation, measurements are taken via calls to the measurement functions, using Parasol's foreign function interface. At the end of simulation, the objective function uses these measurements to score the concrete program. The search algorithm then uses the historical series of scores to select new concrete values for the next iteration. This process repeats for a set time budget. At the end, the optimizer returns the highest-ranked concrete program that successfully compiles to the underlying hardware.

6.4.1 Measurements and Objectives

The objective function to be optimized is written in Python by the user, along with any required extern functions for measurement. These measurements can target any any part of the operating environment: the interpreter prints a trace of each event it handles, allowing for large-scale measurements of network behavior, while extern

```
1 hits = 0
2 misses = 0
3 def logHits(found):
4   global hits, misses
5   if found:  hits += 1
6   else  :  misses +=1
7 def objective():
8   return misses/(hits+misses)
```

Figure 6.5: Measurement and objective functions for the data-plane cache.

functions can be used to log information from within an event handler. Example objective functions include the distribution of flows across paths in a load-balancing application, the rate of collisions in a hash table, and the comparison of a CDF[4] created from run-time measurements to a ground-truth CDF. The search algorithm treats the objective function as a black box; any metric is acceptable.

Objective and measurement functions are often simple. For our data-plane cache, the goal is to minimize the miss rate (that is, the ratio of cache misses to cache accesses). The functions for measuring and computing miss rate can be defined in just eight lines of Python (Figure 6.5).

The measurement function `logHits` is provided as an extern to the Parasol program, and is called once per request, as shown in Figure 6.3. The `objective` function is called by the optimization algorithm at the end of simulation. The global variables `hits` and `misses` are maintained in a single instance of the Python interpreter, so their values persist throughout the execution of the program.

Comparing with ideal implementations. A particularly useful type of measurement is to compare the runtime behavior of a data structure against an idealized implementation. As an example, a data-plane application can produce round-trip time (RTT) samples by matching SYN packets with corresponding SYN-ACKs [24, 68].

[4]Cumulative Distribution Function.

When the switch sees a SYN packet, it stores its timestamp in memory, and can compute the RTT when it sees the corresponding SYN-ACK packet. However, if the data structure is full, the switch cannot store new SYN packets; as a result, the application can only provide a portion of RTT measurements. During simulation, a measurement function could maintain a Python data structure which does not run out of memory, and compare its results to those of the Parasol structure—this provides an easy-to-compute ground truth for how well the Parasol program could possibly perform.

6.4.2 Search Algorithm

The final component of the Parasol optimizer is the search algorithm itself. The goal of the search algorithm is to find parameter values that minimize the objective function. However, the space of possible solutions can be intractably large. Worse, many of these solutions may not even compile to hardware, even if they give good results in the interpreter! Doing an exhaustive search is inefficient, and a naïve strategy may never discover a compiling solution.

As a strawman solution, Parasol could require users to restrict the search space by providing bounds on all variables. However, this will almost certainly include a large number of non-compiling solutions, as even experts would have trouble determining appropriate bounds. As an example, reasonable bounds on cache with a CMS as the key tracker might be 1-5 cache tables and CMS rows, and while restricting cache entries and CMS columns fit within a single stage. These bounds produce a solution space of 4225 configurations, only 16% of which actually compiled to the switch during testing.

Alternatively, Parasol could use a heuristic to exclude non-compiling solutions before they are simulated, assigning them a maximum cost. While this avoids simulating configurations that do not compile, it also reduces the effectiveness of the

search strategies because non-compiling solutions do not give any indication of how to vary their parameters to become compilable. One could imagine simulating the configuration anyway, in the hopes that it will lead us to a compiling configuration, but this is unlikely – programs using an impossible amount of memory, for example, are likely to perform impossibly well.

In practice, we address this issue by splitting the search algorithm into two phases: preprocessing and simulation. In the first phase, Parasol automatically prunes non-compiling solutions from the search space, without requiring user-defined bounds. In the second phase, Parasol searches the space of remaining solutions with a user-configurable search algorithm.

Preprocessing In a nutshell, the goal of the preprocessing phase is to identify solutions that are making maximal use of the resources on the switch, without using so many that the program fails to compile. Solutions which do not fully exploit the switch's resources are likely to be outperformed by those that do. The resources we consider are memory, pipeline stages, hash units, array accesses, and ALU usage. During this phase we only consider symbolic values that affect resource allocation; non-resource symbolics, such as timeouts or thresholds, are ignored.

We assume that the symbolics are monotonic with respect to resources – that is, increasing the value of any symbolic value should not decrease the amount of resources used. In our experience, this is a safe assumption; certainly, all of the applications we evaluated satisfied this property.

We search for maximal-resource programs by determining upper bounds for each resource symbolic. We begin by setting all symbolic values to either a default or user-provided starting value. We then pick a symbolic, and determine an upper bound for it by iteratively increasing only that symbolic's value until we run out of resources[5].

[5]As measured by a heuristic for whether compilation will succeed. These heuristics are discussed in §6.4.3.

Thanks to monotonicity, the largest value that fits provides an upper bound for that symbolic.

We then pick another symbolic and repeat this process; however, this time we find one upper bound for *each* possible value of the first symbolic. We do the same for the next symbolic, and the next, each time finding an upper bound for all valid combinations of previously processed symbolics. When we finish, we will have enumerated the entire useful search space (i.e., every compiling solution).

Default values The number of solutions we enumerate grows multiplicatively with the number of parameters. To make preprocessing it more tractable, we provide reasonable default values that, in our experience, allow the preprocessor to skip solutions that obviously do not fully exploit the switch's resources. In particular, values that represent memory usage are initially set to the max memory available in a stage; other symbolic values start at 4. The "magic value" of 4 was chosen empirically; we found it generalized well to all of our applications, providing a significant reduction in preprocessing time when compared to the smallest starting value of 1. For example, the preprocessing time for a Precision-based cache improved from almost 2 hours to only 25 minutes.

Simulation In the second phase of the search algorithm, we perform a configurable search through the pruned space of solutions we created during the preprocessing phase. We choose a configuration from that phase, select values for any non-resource symbolics, and execute the resulting program in the Lucid interpreter. We then score the configuration based on its output, and use a customizable search strategy to select the next configuration to evaluate based the history of scores.

The Parasol optimizer is built to accommodate a variety of search algorithms. We provide four built-in search functions for programmers to use – exhaustive search,

Nelder-Mead simplex method, simulated annealing, and Bayesian optimization[6] – but Parasol also supports any programmer-defined search of the solution space, and is compatible with any optimization technique written in Python (e.g., stochastic gradient descent, genetic algorithms, etc.). Programmers are free to choose or write a search algorithm that provides their preferred balance between search time and optimality of the final result. We evaluate the effectiveness of each built-in strategy in §6.5.

6.4.3 Design Tradeoffs

Accelerating Preprocessing The preprocessing phase requires us to analyze the resource usage of a program to determine if it will compile or not. The simplest way to do this would be to actually compile the program; however, compilation can be very slow (the Conquest [23] application took over 13 minutes), and most applications require compiling many configurations (Conquest has a compiling search space of 25 configurations). Instead, we have tested a range of heuristics, with varying trade-offs between performance and accuracy.

We consider three heuristics, all of which operate by emulating the scheduling phase of the Lucid's P4 compiler (§5.5). The primary distinction between the heuristics is the types of resources they account for during scheduling. The simplest heuristic, **dataflow graph**, only accounts for dependencies between actions (two actions cannot be in the same stage if one depends on the output of the other). The **greedy layout** heuristic additionally considers the layout of memory, hash units, array accesses, and ALU usage (for example, we cannot have multiple concurrent accesses to the same array).

The **partial compilation** heuristic actually runs the Lucid compiler to produce a

[6]We chose these strategies because (with the exception of exhaustive) they use the history of scores to efficiently navigate the search space. They also provide a range from simple strategies (exhaustive, Nelder-Mead simplex) to more complex (Bayesian).

Heuristic	Avg compile time	Reduction
Dataflow graph	51s	–
Greedy layout	51s	13%
Partial compilation	1.5min	13%
Full compilation	1.5min	16%

Figure 6.6: The performance of each preprocessing heuristic for a single configuration, averaged over each evaluated application. The greedy layout provides the best balance between performance and accuracy.

P4 program. This is much faster than a full compilation to the Tofino, and additionally considers resource limits on physical tables in the pipeline (such as match column width, maximum table size, and number of actions per stage). The only constraints that we encountered that were not modeled by the Lucid compiler are packet header vector (PHV) clustering constraints – each packet header or metadata variable in a program must be placed into a specific PHV cluster, and each cluster has a fixed number of ALUs in each pipeline stage. In our experience, it was possible to run afoul of PHV constraints in sufficiently complicated programs, but these violations were unaffected by choice of parameter values. Our preliminary implementations of 6/10 applications failed to compile with *any* configuration due to PHV constraints, but once we adjusted the programs to accommodate for the constraints, we did not run into PHV constraint violations for any configurations.

We summarize the performance of our heuristics in Figure 6.6. We list the average compile time for each of our evaluated applications and the average reduction in search space size, using the dataflow graph heuristic as the baseline. In practice, we have found that the greedy layout heuristic provides the best tradeoff between performance and accuracy. We cope with the potential inaccuracy of the heuristic by including a safeguard to ensure that Parasol returns a compiling solution. Specifically, we actually compile the highest-ranked configuration at the end of our optimization loop. Should compilation fail, Parasol tries the next-highest-ranked, and so on, until

one compiles. If none of the tested solutions compile, the system will repeat the optimization process, excluding solutions that did not compile.

We found that in practice, this rarely happens. After manually fixing any PHV errors, the optimal solutions for nine out of the ten applications fit within the target resources. Only one of the applications (CMS) resulted in an "optimal" configuration that did not compile; however, the Parasol optimizer quickly found a compiling solution that had similar performance.

Trace Representativity Since the Parasol optimization framework is simulation-based, it relies on receiving a traffic trace that is representative of the target network's conditions. If the actual traffic in the network deviates from the patterns in the trace, the performance of the application may not match the simulated performance. However, because Parasol preserves the semantics of the data-plane program, it will never produce unexpected or invalid behavior—its performance may simply be poorer than anticipated.

To mitigate poor performance, programmers can use multiple traffic traces to optimize their application, and use a weighted combination of performance on the traces as the objective function. We show an example with our data-plane cache in §6.5.4, by optimizing with workloads of different distributions. Alternatively, if the distribution varies in a regular way (e.g. depending on time of day), the programmer can use traces from peak times, where applications are likely most sensitive to poor performance.

Beyond poor performance, an unrepresentative trace can leave an application vulnerable to attacks when the training trace only contains benign traffic. To use Parasol for tuning a security system, one needs traces containing the kinds of attacks the application seeks to detect or prevent. Fortunately, Parasol users need not acquire and label such traces themselves, as the network security community already goes to

great lengths to produce and share traces for the evaluation of their own security systems [13]. These traces come from a variety of sources, including cyber defense exercises [20] and security-oriented testbeds or simulators [80, 21].

6.5 Evaluation

Our evaluation of Parasol addresses its two components:

- **Language**: Can Parasol express a wide variety of parameters, objective functions, and data-plane applications?
- **Optimizer**: How well do optimized Parasol programs perform, and how quickly does Parasol find good parameters?

To answer these questions, we used Parasol to implement and optimize a suite of ten data-plane applications (shown in Figure 6.7) with respect to representative traffic traces. We chose applications that encompass a wide array of structures (including commonly used structures like sketches and hash tables) and contain a diverse set of parameters and objective functions. The code implementing Parasol is integrated into the open-source Lucid repository on GitHub.[7].

In the remainder of the section, we discuss each Parasol component individually, and finish with an in-depth look at our data-plane caching example. We used three types of traces in our evaluation – the University of Wisconsin Data Center Measurement trace [11], a trace from core Internet routers [18], and synthetic traces for the cache application. When evaluating applications, we split out our traces into training and testing data sets. To help mitigate the risk of overfitting, we optimize our programs on the training data set, and then measure its performance on the testing data.

[7] `https://github.com/PrincetonUniversity/lucid`

	Parameter Classes				
Application	**MA**	**TH**	**DS**	**TM**	**Objective (LoC)**
Count-min sketch (CMS)	√				Mean estimate Error (20)
Multi-hash table (MHT)	√				Collision ratio (11)
Data plane cache (KV [75])	√	√	√	√	Miss rate (23)
RTT monitor (RTT [24])	√			√	Read success rate (118)
Unbiased RTT (Fridge [89])	√	√			Max percentile error (88)
Starflow [74]	√				Eviction ratio (17)
Conquest [23]	√				F-score (101)
load balancing (LB [78])	√	√			Error vs. optimal (38)
Precision [10]	√				Avg. error for top flows (28)
Stateful Firewall (SFW [73])	√	√		√	Packet overhead (70)

Figure 6.7: Applications optimized with Parasol, showing which classes of parameters/objective functions were used. The four classes of parameters are Memory Allocation (MA), Thesholds (TH), Data Structure Selection (DS), and Timing (TM).

6.5.1 Language

To optimize a program using Parasol one must be able to do three things: write the program in Lucid, represent the parameters of interest with symbolic values, and write an appropriate objective function. Accordingly, to measure the expressivity of Parasol, we implemented several applications with multiple classes of parameters and diverse objectives. These programs are listed in Figure 6.7. Figure 6.8 shows the high-level benefit of Parasol over the existing frameworks for application-level parameter optimization, P4All [36] and SketchGuide [91]. While Parasol allowed us to fully express the optimization goal of each application (both parameters and objective function), P4All and SketchGuide could only express the full optimization goals of 2/10 applications. In the rest of this section, we discuss the ability of Parasol to represent a diversity of both parameters (its "program flex"), and objective functions.

Program Flex. As Figure 6.7 shows, the Parasol programs we implemented had four general classes of parameters: memory allocation, decision thresholds, choice of data structure, and operation timing. These classes encompassed a diverse range of

Application	Params	Objective
Count-min sketch (CMS)	√	√
Multi-hash table (MHT)	√	√
Data plane cache (KV [75])	✗	✗
RTT monitor (RTT [24])	✗	✗
Unbiased RTT (Fridge [89])	✗	√
Starflow [74]	√	✗
Conquest [23]	√	✗
load balancing (LB [78])	✗	✗
Precision [10]	√	✗
Stateful Firewall (SFW [73])	✗	✗

Figure 6.8: Applications optimized with Parasol, showing which parameters and objective functions could be fully modeled by either P4All or SketchGuide.

parameters, including data structure size, time between packets, heavy hitter threshold, and probability of an item being added to a structure. The generality of symbolic values allowed Parasol to express all of them.

In comparison, P4All and SketchGuide could only support parameters from 5/10 of our implemented applications (CMS, MHT, Starflow, Conquest, Precision) as it is impossible to express threshold, timing, or data structure choice parameters in P4All or SketchGuide. Even for the examples that *could* potentially be optimized by P4All or SketchGuide, it is easy to imagine slightly more complex variants that would require incompatible parameters. For example, our CMS is a simple implementation with no concept of time intervals – it never resets. Most applications, however, will want to count over intervals, which requires a mechanism to periodically reset or age counters, and a parameter that controls the length of the interval. The addition of that one simple parameter makes the "deployable" variant of CMS incompatible with P4All and SketchGuide.

Objective functions. The objective functions for our applications measured a wide variety of high-level properties (Figure 6.7). These functions were generally short and simple: on average, each function was approximately 50 lines of Python code. The

only requirement for Parasol objective functions is that they be expressible in Python. They can take as input any, all, or none of the parameters of the application, along with any measurements taken during the simulation.

In contrast, existing systems (P4All, SketchGuide) require programmers to supply a closed-form objective function, which specifies exactly how the parameters relate to the final cost. In practice, this can be very difficult, particularly for applications that do not have theoretical guidelines or proven error bounds. Such applications are common; even in research, many data-plane applications are evaluated empirically, without finding provable theoretical guarantees [74, 23]. Furthermore, many systems are composed of multiple components or data structures; writing a closed-form function for those systems requires not just understanding each component individually, but codifying precisely how they interact.

In our evaluation, we considered an objective function to be expressible in P4All or SketchGuide only if we could find a derivation in existing literature that included all the parameters of the application, even if those parameters were not themselves expressible in P4All or SketchGuide. We consider deriving a closed-form objective function to be beyond the scope of an application developer (and also this thesis) as it requires significant theoretical work.

With these criteria, we found that we were only able to express three out of our ten objective functions in P4All or SketchGuide. Even so, there is a caveat: functions from the literature typically quantify worst-case performance. These objective functions oftentimes do not provide a realistic idea of how the application performs in practice, and applications optimized for the worst case may not perform as well on practical workloads. In contrast, Parasol objective functions measure actual performance on a sample trace, and are therefore able to optimize for a much broader range of criteria, even when a closed-form error function exists [91, 54, 22].

6.5.2 Optimization Quality

We evaluate the quality of Parasol's solutions, compared to both hand-optimized systems and an oracle optimizer (described below), and analyze the factors that impact it. All experiments in this section use a two-hour time limit for the dynamic search phase of the Parasol optimizer.

First, we compare the results of optimization with Parasol to optimization with an "oracle". Whereas the Parasol optimizer chooses its parameters via a search on its training data, the oracle optimizer chooses parameters by exhaustively searching the testing data set, i.e., it always chooses the optimal parameters.

Parasol found configurations that performed as well as the oracle for 6/10 applications (CMS, MHT, RTT, Starflow, Precision, and SFW). For 3/10 applications (KV, Fridge, Conquest), the relative difference between the objective score of Parasol's and the oracle's configuration (i.e., $\frac{|\text{Objective}_{\text{oracle}} - \text{Objective}_{\text{Parasol}}|}{\text{Objective}_{\text{oracle}}}$) was under 1.1%. For the remaining application, the load balancer (LB), Parasol's solution was, in relative terms, 82% worse than the oracle. However, in absolute terms the difference was small: the oracle's configuration performed 1.7% worse than a perfect load balancer, while Parasol's configuration performed 3.1% worse than a perfect load balancer.

The Parasol Preprocessor

To measure the effect that Parasol's preprocessor had on the solution quality, we compared application performance when optimized with and without preprocessing, using the same two-hour time budget for Parasol's search phase. When the preprocessor was disabled, we bounded the search space by setting the same initial bounds for all memory allocation variables — 20 register arrays and the max amount of SRAM per stage for registers. These constitute reasonable bounds in our judgement – high enough to include all compiling solutions for each application without unnecessarily inflating the search space. Additionally, without the preprocessor, we assigned a pre-

determined max cost to solutions that did not compile (e.g., 100% cache miss rate), to avoid delving into unusable areas of the search space.

As shown below, preprocessing consistently improved the performance of the final solution, especially for applications that had a large search space or used multiple structures that compete for resources (Starflow, KV, Conquest, SFW). In fact, when the cache used CMS as the key tracker, Parasol consistently did not find a compiling solution in the time budget without preprocessing. The results for the most sophisticated applications were:

- For Conquest, enabling the preprocessor improved recall from 75% to 87%.
- For Starflow, the preprocessor improved eviction ratio from 35% to 15%.
- For the stateful firewall, the preprocessor improved recirculation and retransmission overhead from 16 kbps to 0.01 kbps.

Applications that had a small search space (CMS, MHT, Fridge, LB) did not perform significantly better when preprocessing was enabled. However, even for such applications, preprocessing still has an important benefit: it automatically bounds the search space for the programmer, without the need for them to manually "guess" reasonable bounds.

The Parasol Searcher

We found that the effectiveness of Parasol's search phase depended on two factors: the search strategy and the quality of the input trace. Parasol provides four built-in strategies: exhaustive search, Bayesian, simulated annealing, and Nelder-Mead simplex. We note that all of these strategies (except exhaustive) have hyperparameters that control the learning process. We chose hyperparameter values manually such that strategies produce solutions as good as or near the oracle solutions. We found

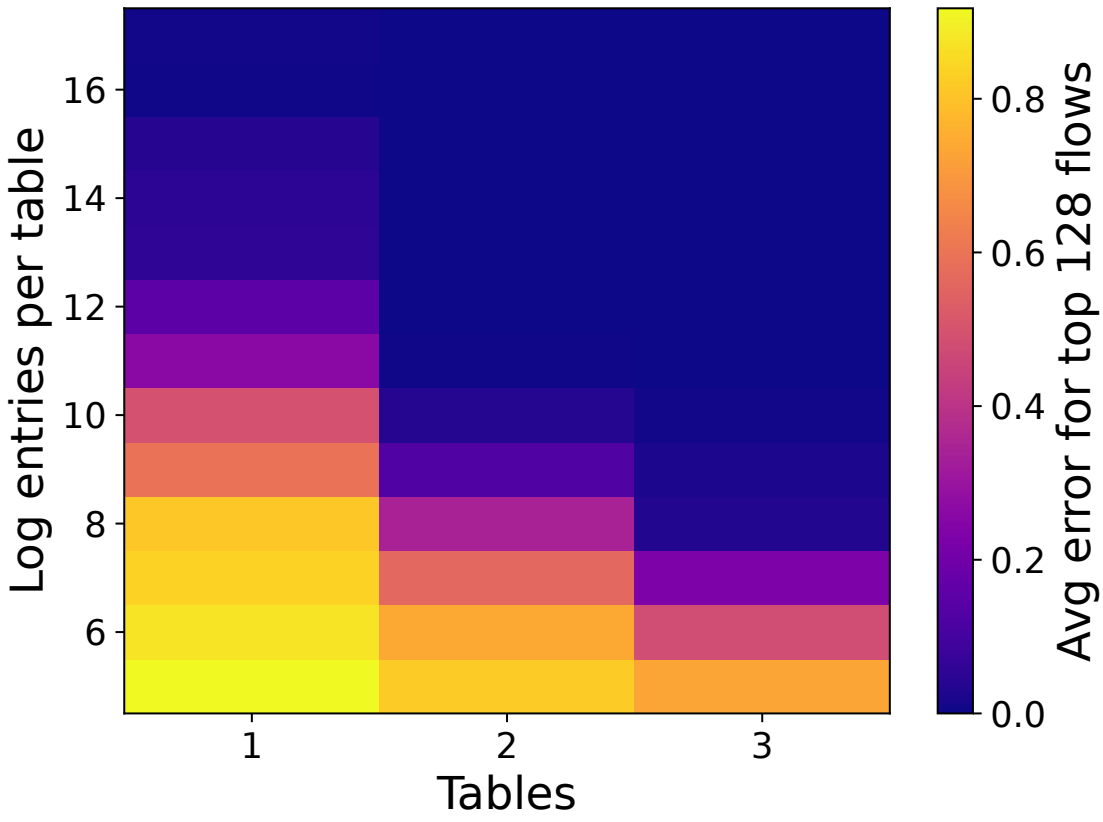

Figure 6.9: A graphical representation of the search space for the Precision application. A darker color represents a lower error. The optimal configuration achieved an error of 0.01%, and nearly 40% of the solution space produced an error of less than 1%.

that we could re-use these values for all applications without negatively affecting solution quality.

Search strategy For some applications, the choice of search strategy does not matter because a large portion of the compiling solution space is near-optimal. For example, in the Precision application, over half of the search space after preprocessing contained solutions that produced an average error of less than 10% (Figure 6.9), compared to the optimal of less than 1%. In such cases, the search methods mostly converged to the same configuration or to configurations that had very similar performance.

For more complex applications, we found that no single search strategy dominated. Because of this, we found that the best tactic was to run multiple search strategies in parallel for each application, and choose the best result from among them. Conversely, for applications with a small search space (after preprocessing), we simply used exhaustive search. We consider a search space to be small if the exhaustive search completed within the two-hour time budget.

Training trace. The effectiveness of Parasol's search phase depended on the training trace's size and representativeness.

Across all applications, we found that traces with approximately 1 million packets were sufficiently large for Parasol to find high quality (i.e., near optimal) configurations. Training trace size mattered more for some applications than others. In particular, when optimizing a hash table, the trace has to be large enough to cause hash collisions; otherwise the differences between configurations are small and it is difficult (or impossible) for Parasol's search algorithm to find the best one. For example, the Starflow configurations found by the simplex and Bayesian strategies resulted in similar eviction ratios (12% and 5%, respectively) in a small trace of 5000 packets, but had very different errors (46%, 26%) with a larger trace of 5 million packets.

In addition to size, the trace's representivity was also important. For some applications, the search phase was only effective when a trace contained certain network events. For example, the Conquest data structure detects microbursts, and only begins monitoring when one occurs. A trace with no microbursts would produce no meaningful objective values, regardless of the configuration. On the other hand, some applications were less sensitive to differences between training traces and target workloads. When testing Starflow, we found that Parasol was able to find near-optimal solutions for a Wide-Area Network (WAN) using training traces from either a WAN or a datacenter.

Comparison to hand-optimized configurations

We compared the performance of Parasol configurations to that of hand-tuned configurations for our three most complex applications: Fridge, Conquest, and Starflow. The hand-tuned configurations come from the applications' original evaluations [89, 23, 74]. Our goal is to determine whether Parasol can essentially reproduce these results, by finding configurations that perform comparably on a similar workload.

The case studies are described in detail in Appendix B.1. At a high level, Parasol solutions performed reasonably close to the hand-optimized solutions for all three applications.

- For Fridge, Parasol found a configuration that achieved a delay estimation error of 18%, compared with the original evaluation's result of 25%.
- For Conquest, Parasol found a configuration with a precision of 97% and recall of 87%, compared to the original evaluation which found precision and recall $>$ 90%, using the same trace.
- For Starflow, Parasol found a configuration with an eviction ratio of 15% in a wide area workload, which is better than the 18% eviction ratio reported in the original evaluation.

6.5.3 Optimizer Speed

The runtime of the Parasol optimizer (shown in Figure 6.10) is application-dependent, and has two major components: preprocessing time and search time. Preprocessing time scales with the complexity of the input program and number of parameters, and took between 2 seconds and 1.5 hours. Search time scales primarily with the size of the input trace, and was limited to 2 hours, though many applications required

App	Preprocess time	Training size	Training time	Testing size	Testing time
CMS	16s	500k	25s	10M	12min
MHT	15s	1M	47s	10M	7min
KV+Precision	25min	1M	6min	5M	25min
KV+CMS	2hrs	1M	2min	5M	7min
RTT	23s	1M	1min	3M	3min
Fridge	3s	1M	56s	3M	2min
Starflow	1.5hr	900k	1min	5M	27min
Conquest	15s	10M	9min	10M	10min
LB	2s	500k	16s	3M	2min
Precision	32min	1M	6min	18M	1.7hrs
SFW	30s	4M	3min	11M	7min

Figure 6.10: Runtime of Parasol components per application. Preprocess time is the total time to preprocess the program using the greedy layout heuristic. Testing/Training size number of packets in the respective trace, and Testing/Training time the average time to simulate the trace once for an application.

less than that. A single iteration of the training trace took between 16 seconds to 9 minutes, depending on application.

Overall, the Parasol optimizer took no more than 3.5 hours to find near-optimal settings for any of our applications. This compares favorably to compiling, testing, and tuning applications by hand: just compiling *one* configuration of a program to a reconfigurable architecture like the Tofino can take hours [32] for both research or industrial compilers, because it is a fundamentally hard task [79].

As mentioned above, we found that three main factors influenced the overall runtime: application complexity, training set size, and search strategy.

Application complexity. The preprocessing time depends on the complexity of the program, both in terms of length and number of parameters. Programs with more parameters (e.g. Starflow) took longer than programs with few parameters (e.g. LB). Figure 6.10 lists the total preprocessing time for each application.

Complex programs also take longer to simulate. The CMS simulation took about

a minute for a 1 million packet trace, while a trace of the same size with Precision took three minutes. Precision is more complex because it contains logic for recirculating packets, while the CMS does not recirculate packets. The recirculation not only adds complexity to Precision, it also requires the program to process more packets, as each recirculation creates a new packet that must be processed.

Training set size. The runtime of Parasol's search phase increases roughly linearly with the size of the input training trace, because the search algorithm executes the trace once for each candidate configuration. Reducing the size of the provided trace can speed up optimization, but many applications require large traces. For example, evaluating the performance of a program that measures heavy hitters (e.g., Precision) requires enough traffic that the trace contains heavy flows.

Search strategy. Search strategies took different amounts of time to converge, depending on the application. As a point of comparison, we evaluated the load balancing and Starflow applications by tracking the best evaluated configuration after *each* iteration.

For the load balancer, all three methods found similarly performing configurations, but the overall search time was much different: Bayesian search took approximately 19 minutes, while simulated annealing and simplex search took only 2 minutes. For Starflow, the Bayesian and simulated annealing strategies reached configurations with similar performance (in 13 and 10 minutes, respectively) while simplex did not find a configuration that produced the best collision rate within the time budget. Overall, there is no clear "best search strategy" that works across applications.

6.5.4 Case Study: Data-plane Caching

To better understand how Parasol handles workload dependence and how the distribution affects the performance of different structures, we provide a more detailed

look at our original example, the cache application. We optimize the cache for three different workloads: skewed Zipfian (top 10 keys had 58% of requests), less skewed Zipfian (top 10 keys had 15% of requests), and uniform (top 10 keys had .06% of requests). Training traces contained 1 million requests, and test traces contained 5 million requests. We limit the cache size to 10K entries.

We first compare a cache with a CMS key tracker (NetCache [41]) to one using a Precision key tracker. The skewness of the workload significantly impacted cache performance. For the skewed trace, the Precision cache slightly outperformed CMS (7% vs. 10% miss rate, respectively). For the less skewed trace, Precision again slightly outperformed CMS (64% vs. 69% miss rate, respectively). However, they achieved the same miss rate for the uniform distribution (73% miss rate). The uniform distribution puts more pressure on the key tracker because there are more unique keys, causing worse performance than the skewed workload.

Both the Precision and CMS key trackers are complex applications that require recirculation to insert keys. As an alternative, we implemented a single-stage hash table as a cache, with no key tracker. The hash table always evicts keys on collisions, and thus does not require recirculation. For skewed and uniform workloads, the hash table performed similarly to Precision and CMS (10% skew, 73% uniform miss rate). Given that more complex structures provide marginal benefit, the hash version might be preferred for these workloads because of its simplicity. However, for the less skewed trace, the performance of the hash cache deviated more noticeably (70% miss rate). For this distribution, the added complexity of Precision provides a more notable improvement.

Even though the hash table had similar performance to CMS as a key tracker, CMS has an important benefit: the CMS only evicts items when requests for an uncached key reaches a threshold, whereas the hash table will always replace on collision. In other words, the items stored in the cache are more rapidly changing in

the hash table.

Overfitting To mitigate overfitting, we also optimized our cache using a combination of the three traces. Our objective function was the average of the miss rates for each trace. The layouts chosen by Parasol for the Precision and hash versions were the same as when training with each workload individually, with Precision providing the lowest miss rate. The layouts chosen by Parasol for the CMS key tracker differed when trained on all three distributions. They performed slightly worse on each distribution individually — achieving miss rates that were approximately 1% worse for each distribution.

Parasol has the flexibility to express arbitrary programs with arbitrary parameters, that can be optimized with any objective, for any workload. With Parasol, we can directly compare different caching structures, for multiple traffic distributions, by simply tweaking a boolean value. In doing so, we found that the structure used in literature (CMS), is not always the best structure to track keys in a data-plane cache. Parasol is able to make this process easy because it lifts the burden of reasoning about how parameter choices can affect performance off of the programmer, greatly simplifying the development process for data-plane applications.

6.6 Related Work

Researchers have developed a number of programming and synthesis tools to more easily write data-plane programs. Domino [70], Chipmunk [32], Lyra [31], and O4 [4] provide new, high-level languages for expressing data-plane programs, each with abstractions and a compiler targeting one or more architectures. These compilers include optimizations or synthesis techniques to ensure that programs compile. However, if a program cannot fit on a target, the program will not compile. They also do not

provide environment-specific optimizations, as the compiler does not have access to traffic information.

There also exist tools for optimizing prewritten data-plane programs. P^{2}GO [83] uses a traffic trace to minimize the resources used by a P4 program by reducing dependencies that do not appear in practice, shrinking tables, and offloading parts of the program to a controller. Cetus [49] uses static analysis to eliminate dependencies between tables and to merge tables. Although P^{2}GO and Cetus can fit programs into limited resources, they either do not provide environment optimizations or risk changing program semantics.

A third type of tool optimizes by leveraging user domain knowledge. P5 [3] uses a high-level description of the network's policy to remove spurious dependencies and unused features. P4All [36] and SketchGuide [91] allow users to declare flexibly sized structures and optimize them with a user-provided objective function. By taking policy into account, these tools can provide more detailed optimizations than would otherwise be possible. However, they ask a lot of their users; P5 requires a high-level policy description, and P4All and SketchGuide require a closed-form objective function.

An area of work related to the implementation of Parasol's optimizer is network simulation. Network simulators are designed for many objectives, including high fidelity [65], interactive operation [48], automatic traffic generation [87], and scalable performance [82]. In general, all of these tools complement Parasol. Future work on Parasol will likely involve integrating these tools to improve the capabilities, fidelity, and performance of Parasol's simulator.

Chapter 7

Conclusion

Modern programmable networking hardware provides network operators and academics with unprecedented ability to control the behavior of their networks. However, simply having the hardware is not enough: operators must also be able to actually *write* the programs they care about. Writing sophisticated programs with modern tools is an arduous task; to compensate, this thesis has introduced the Lucid language as an easy-to-use, correct-by-construction dataplane programming language.

In contrast to existing languages, Lucid provides a high-level event-based view of the network. Different logical threads of control are naturally represented as different events; users do not need to worry about their interaction in the hardware, since the Lucid compiler interleaves them automatically. The abstraction of events spares users from the burden of manually configuring their hardware for each possible execution path.

In addition to its simple programming model, Lucid makes it easy to write correct code by providing simple, high-level guidelines for what correct code should look like. These guidelines are automatically enforced by the type system and other syntactic checks, providing users with useful feedback when they are violated.

The Lucid language has proven itself to be not only a good way of expressing

programs, but also an excellent substrate for compiler passes. The Lucid compiler breaks down programs into their atomic components, then incrementally merges them while computing a layout on the Tofino. Doing so allows users to write substantially more sophisticated programs than they could relying on the Tofino compiler alone. Although the compiler operates using greedy algorithms, it has a very high success rate of fitting programs into the available hardware resources.

Finally, Lucid can be used for more than simply writing programs. The Parasol framework provides a way to optimize the *high-level* behavior of arbitrary Lucid programs, with unprecedented flexibility in the types of objectives and the parts of the program that can be optimized.

Lucid has already proven itself an easy-to-use and flexible substrate for data-plane programming. Many of its ideas, such as its pipeline type system, are reusable and might be applied to other languages, or even other domains in the future. Furthermore, although Lucid is currently bound to the Tofino, we hope to extend its capabilities to other targets in the future, such as SmartNICs or EBPF. With Lucid already in use for multiple research projects, we hope that in the future it will be a staple of network programming for academics and network operators alike.

Appendix A

Pipeline Types

A.1 Operational Semantics

PAIR-1

$$\frac{M, z, e_1 \to M', z', e_1'}{M, z, (e_1, e_2) \to M', z', (e_1', e_2)}$$

PAIR-2

$$\frac{}{M, z, (v_1, e_2) \to M', z', (v_1, e_2')}$$

FST-1

$$\frac{M, z, e \to M', z', e'}{M, z, \texttt{fst}\ e \to M', z', \texttt{fst}\ e'}$$

FST-2

$$\frac{}{M, z, \texttt{fst}\ (v_1, v_2) \to M', z', v_1}$$

SND-1

$$\frac{M, z, e \to M', z', e'}{M, z, \texttt{snd}\ e \to M', z', \texttt{snd}\ e'}$$

SND-2

$$\frac{}{M, z, \texttt{snd}\ (v_1, v_2) \to M', z', v_2}$$

LET-1

$$\frac{M, z, e_1 \to M', z', e_1'}{M, z, \texttt{let}\ id = e_1\ \texttt{in}\ e_2 \to M', z', \texttt{let}\ id = e_1'\ \texttt{in}\ e_2}$$

LET-2

$$\frac{}{M, z, \texttt{let}\ id = v\ \texttt{in}\ e \to M, z, e[v/id]}$$

DEREF-1

$$\frac{M, z, e \to M', z', e'}{M, z, !e \to M', z', !e'}$$

DEREF-2

$$\frac{z \leq z_e}{M, z, !\texttt{addr}(z_e) \to M, S(z_e), M[z_e]}$$

UPDATE-1

$$\frac{M, z, e_1 \to M', z', e_1'}{M, z, e_1 := e_2 \to M', z', e_1' := e_2}$$

UPDATE-2

$$\frac{M, z, e \to M', z', e'}{M, z, v := e \to M', z', v := e'}$$

UPDATE-3

$$\frac{z \leq z_e}{M, z, \texttt{addr}(z_e) := v \to M[z_e := v], S(z_e), ()}$$

IF-1

$$\frac{M, z, e_1 \to M', z', e_1'}{M, z, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to M', z', \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

IF-TRUE

$$\frac{}{M, z, \texttt{if true then } e_2 \texttt{ else } e_3 \to M, z, e_2}$$

IF-FALSE

$$\frac{}{M, z, \texttt{if false then } e_2 \texttt{ else } e_3 \to M, z, e_3}$$

VECTOR

$$\frac{M, z, e_0 \to M', z', e_0'}{M, z, \texttt{vector}(v_0, \ldots, v_n, e_0, \ldots, e_m) \to M', z', \texttt{vector}(v_0, \ldots, v_n, e_0', \ldots, e_m)}$$

INDEX-1

$$\frac{M, z, e \to M', z', e'}{M, z, e[n] \to M', z', e'[n]}$$

INDEX-2

$$\frac{n \leq m}{M, z, \texttt{vector}(v_0, \ldots, v_m)[n] \to M, z, v_n}$$

LOOP

$$\frac{}{M, z, \texttt{for}\ b < n\ \texttt{do}\ e \to M, z, e[0/b]; ...; e[n-1/b]; ()}$$

COMP

$$\frac{}{M, z, [e\ \texttt{for}\ b < n] \to M, z, \texttt{vector}(e[0/b], \ldots, e[n-1/b])}$$

APP-1

$$\frac{M, z, e_1 \to M', z', e_1'}{M, z, e_1\ [\overline{k}, \overline{\ell}]\ e_2 \to M', z', e_1'\ [\overline{k}, \overline{\ell}]\ e_2}$$

APP-2

$$\frac{M, z, e_2 \to M', z', e_2'}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ e_2 \to M', z', v_1\ [\overline{k}, \overline{\ell}]\ e_2'}$$

APP-3

$$\frac{v_1 = \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}]\ (id : \tau, \ell) \to e_{body}}{M, z, v_1\ [\overline{k}, \overline{\ell}]\ v_2 \to M, z, e_{body}[v_2/id][\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

A.2 Well-formedness conditions

A.2.1 Size rules

$$\frac{}{\Delta, \mathbb{K} \vdash n} \qquad \frac{\kappa \in \Delta}{\Delta, \mathbb{K} \vdash \kappa} \qquad \frac{b \in \mathsf{Dom}(\mathbb{K})}{\Delta, \mathbb{K} \vdash b}$$

A.2.2 Location rules

$$\frac{}{\Delta, \mathbb{K} \vdash 0} \qquad \frac{\alpha \in \Delta}{\Delta, \mathbb{K} \vdash \alpha} \qquad \frac{\Delta, \mathbb{K} \vdash \ell}{\Delta, \mathbb{K} \vdash S(\ell)} \qquad \frac{\Delta, \mathbb{K} \vdash \ell}{\Delta, \mathbb{K} \vdash \ell.0} \qquad \frac{\Delta, \mathbb{K} \vdash \ell \qquad b \in \mathsf{Dom}(\mathbb{K})}{\Delta, \mathbb{K} \vdash \ell.b}$$

A.2.3 Constraint rules

$$\frac{}{\Delta, \mathbb{K} \vdash \texttt{true}} \qquad \frac{\Delta, \mathbb{K} \vdash \ell_1 \qquad \Delta, \mathbb{K} \vdash \ell_2}{\Delta, \mathbb{K} \vdash \ell_1 \leq \ell_2} \qquad \frac{\Delta, \mathbb{K} \vdash C_1 \qquad \Delta, \mathbb{K} \vdash C_2}{\Delta, \mathbb{K} \vdash C_1 \wedge C_2}$$

A.2.4 Type rules

$$\frac{}{\Delta, \mathbb{K} \vdash \texttt{Unit}} \qquad \frac{}{\Delta, \mathbb{K} \vdash \texttt{Bool}} \qquad \frac{}{\Delta, \mathbb{K} \vdash \texttt{addr}(T)} \qquad \frac{\Delta, \mathbb{K} \vdash t_1 \qquad \Delta, \mathbb{K} \vdash t_2}{\Delta, \mathbb{K} \vdash (t_1, t_2)}$$

$$\frac{\Delta, \mathbb{K} \vdash t \qquad \Delta, \mathbb{K} \vdash k}{\Delta, \mathbb{K} \vdash \texttt{vector}(t, k)} \qquad \frac{\Delta, \mathbb{K} \vdash t \qquad \Delta, \mathbb{K} \vdash \ell}{\Delta, \mathbb{K} \vdash t\langle \ell \rangle}$$

$$\frac{\begin{array}{c}\Delta' = \Delta \cup \overline{\kappa} \cup \overline{\alpha} \qquad \Delta', \mathbb{K} \vdash C_f \qquad \Delta', \mathbb{K} \vdash \tau_{in} \qquad \Delta', \mathbb{K} \vdash \ell_{in} \qquad \Delta', \mathbb{K} \vdash \tau_{out} \\ \Delta', \mathbb{K} \vdash \ell_{out} \qquad C_f \Rightarrow \ell_{in} \leq \ell_{out} \qquad \forall \ell_1, \ell_2 \in C_f . C_f \Rightarrow \ell_{in} \leq \ell_1 \leq \ell_2 \leq \ell_{out}\end{array}}{\Delta, \mathbb{K} \vdash \texttt{fun}\ \forall \overline{\kappa}, \overline{\alpha}. C_f \Rightarrow (\tau_{in}, \ell_{in}) \rightarrow (\tau_{out}, \ell_{out})}$$

A.2.5 Environment rules

Definition A global declaration $\mathbb{G}$ is well-formed, written $\vdash \mathbb{G}$, if for any two concrete locations z_1, z_2 where z_1 is a strict prefix of z_2, at most one of $\mathbb{G}[z_1], \mathbb{G}[z_2]$ exists.

$$\frac{}{\Delta \vdash \emptyset} \qquad \frac{\Delta \vdash \mathbb{K} \qquad b \notin \mathsf{Dom}(\mathbb{K}) \qquad \Delta, \mathbb{K} \vdash k}{\Delta \vdash \mathbb{K}, b \leq k} \qquad \frac{\Delta, \mathbb{K} \vdash \Gamma \qquad x \notin \mathsf{Dom}(\Gamma) \qquad \Delta, \mathbb{K} \vdash \tau}{\Delta \vdash \Gamma, x := \tau}$$

$$\frac{\vdash \mathbb{G} \qquad \Delta \vdash \mathbb{K} \qquad \Delta, \mathbb{K} \vdash \Gamma}{\vdash (\mathbb{G}, \Delta, \mathbb{K}, \Gamma)}$$

A.2.6 Typing Judgement

These rules are identical to the ones in §4.2; we repeat them here purely for convenience.

$$\textsc{Unit}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash () : \texttt{Unit}\langle \ell' \rangle, \ell, \texttt{true}} \qquad \textsc{True}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{true} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}}$$

$$\textsc{False}\ \frac{\Omega \vdash \ell'}{\Omega, \ell \vdash \texttt{false} : \texttt{Bool}\langle \ell' \rangle, \ell, \texttt{true}} \qquad \textsc{Addr}\ \frac{\Omega.\mathbb{G}[z] = T}{\Omega, \ell \vdash \texttt{addr}(z) : \texttt{addr}(T)\langle z \rangle, \ell, \texttt{true}}$$

$$\textsc{Var}\ \frac{\Omega.\Gamma[id] = \tau}{\Omega, \ell \vdash id : \tau, \ell, \texttt{true}} \qquad \textsc{Pair}\ \frac{\Omega, \ell_0 \vdash e_1 : t_1\langle \ell.0 \rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : t_2\langle \ell.1 \rangle, \ell_2, C_2}{\Omega, \ell_0 \vdash (e_1, e_2) : (t_1, t_2)\langle \ell \rangle, \ell_2, C_1 \wedge C_2}$$

$$\textsc{Fst}\ \frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{fst}\ e : t_1\langle \ell.0 \rangle, \ell_1, C_1} \qquad \textsc{Snd}\ \frac{\Omega, \ell_0 \vdash e : (t_1, t_2)\langle \ell \rangle, \ell_1, C_1}{\Omega, \ell_0 \vdash \texttt{snd}\ e : t_2\langle \ell.1 \rangle, \ell_1, C_1}$$

$$\textsc{Let}\ \frac{\Omega, \ell_0 \vdash e_1 : \tau_1, \ell_1, C_1 \qquad \Omega.(\Gamma[id := \tau_1]), \ell_1 \vdash e_2 : \tau_2, \ell_2, C_2}{\Omega, \ell_0 \vdash \texttt{let}\ id = e_1\ \texttt{in}\ e_2 : \tau_2, \ell_2, C_1 \wedge C_2}$$

If-left

$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle\ell\rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_2 \leq \ell_3}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_3, C_1 \wedge C_2 \wedge C_3}$$

If-right

$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{Bool}\langle\ell\rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : \tau, \ell_2, C_2 \qquad \Omega, \ell_1 \vdash e_3 : \tau, \ell_3, C_3 \qquad \ell_3 \leq \ell_2}{\Omega, \ell_0 \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, \ell_2, C_1 \wedge C_2 \wedge C_3}$$

Abs

$$\frac{(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \qquad \Delta' = \Omega.\Delta \cup \overline{\kappa} \cup \overline{\alpha} \qquad \Delta', \mathbb{K} \vdash \tau_{in}, \ell_{in} \qquad (\mathbb{G}, \Delta', \mathbb{K}, \Gamma[id := \tau_{in}]), \ell_{in} \vdash e : \tau_{out}, \ell_{out}, C \qquad t_f = \forall\overline{\kappa}, \overline{\alpha}.C \Rightarrow (\tau_{in}, \ell_{in}) \to (\tau_{out}, \ell_{out}) \qquad \Omega \vdash \ell' \qquad \Omega \vdash t_f}{\Omega, \ell \vdash \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}](id : \tau_{in}, \ell_{in}) \to e : t_f\langle\ell'\rangle, \ell, \texttt{true}}$$

App

$$\frac{\Omega \vdash \overline{k}, \overline{\ell} \qquad \Omega, \ell_0 \vdash e_1 : t_f\langle\ell'\rangle, \ell_1, C_1 \qquad t_f = \forall\overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \to (\tau_{out}, \ell_{out}) \qquad \Omega, \ell_1 \vdash e_2 : \tau_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_2, C_2}{\Omega, \ell_0 \vdash e_1\ [\overline{k}, \overline{\ell}]\ e_2 : \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], C_1 \wedge C_2 \wedge C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \wedge \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]}$$

Deref

$$\frac{\Omega, \ell_0 \vdash e : \texttt{addr}(T)\langle\ell_2\rangle, \ell_1, C \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash !e : T\langle\ell'\rangle, S(\ell_2), C \wedge \ell_1 \leq \ell_2}$$

Update

$$\frac{\Omega, \ell_0 \vdash e_1 : \texttt{addr}(T)\langle\ell_3\rangle, \ell_1, C_1 \qquad \Omega, \ell_1 \vdash e_2 : T\langle\ell\rangle, \ell_2, C_2 \qquad \Omega \vdash \ell'}{\Omega, \ell_0 \vdash e_1 := e_2 : \texttt{Unit}\langle\ell'\rangle, S(\ell_3), C_1 \wedge C_2 \wedge \ell_2 \leq \ell_3}$$

VECTOR

$$\frac{\Omega, \ell_0 \vdash e_1 : t\langle \ell_v.0\rangle, \ell_1, C_1 \qquad \cdots \qquad \Omega, \ell_{n-1} \vdash e_n : t\langle \ell_v.(n-1)\rangle, \ell_n, C_n}{\Omega, \ell_0 \vdash \texttt{vector}(e_1, \ldots, e_n) : \texttt{vector}(t, n)\langle \ell_v\rangle, \ell_n, C_1 \wedge \cdots \wedge C_n}$$

INDEX-CONST

$$\frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, n')\langle \ell\rangle, \ell_1, C \qquad n < n'}{\Omega, \ell_0 \vdash e[n] : t\langle \ell.n\rangle, \ell_1, C}$$

INDEX-VAR

$$\frac{\Omega, \ell_0 \vdash e : \texttt{vector}(t, k)\langle \ell\rangle, \ell_1, C \qquad \Omega.\mathbb{K}[b] = k}{\Omega, \ell_0 \vdash e[b] : t\langle \ell.b\rangle, \ell_1, C}$$

LOOP

$$\frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \\ \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : \tau, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash \texttt{for}\ b < k\ \texttt{do}\ e : \texttt{Unit}\langle \ell\rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

COMP

$$\frac{\begin{array}{c}(\mathbb{G}, \Delta, \mathbb{K}, \Gamma) = \Omega \\ \alpha_{start} \notin \Delta \qquad \Omega \vdash k \qquad \mathbb{G}, \Delta, (\mathbb{K}, b < k), \Gamma, \alpha_{start} \vdash e : t\langle \ell_v.b\rangle, \ell_{end}, C \\ \texttt{nri}(C, b) \qquad C_0 = C[\ell_{init}/\alpha_{start}][0/b] \qquad \ell_1 = \ell_{end}[\ell_{init}/\alpha_{start}][0/b] \\ C_1 = C[\ell_1/\alpha_{start}][1/b] \qquad \ell_2 = \ell_{end}[\ell_{init}/\alpha_{start}][1/b] \qquad C_2 = C[\ell_2/\alpha_{start}][2/b]\end{array}}{\Omega, \ell_{init} \vdash [e\ \texttt{for}\ b < k] : \texttt{vector}(t, k)\langle \ell_v\rangle, \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b), C_0 \wedge C_1 \wedge C_2}$$

A.3 Properties of Pipe

In this section, we show a variety of properties about Pipe's type system, which we use to prove soundness in appendix A.4. Rather than write out the entire proof of each, we highlight the interesting cases – cases which do not appear are either straightforward or analogous to one of the written cases.

A.3.1 Value Lemmas

These are all proved by inversion or induction on the typing relation.

Lemma (Value Lemma): If $\Omega, \ell \vdash v : \tau, \ell', C$, then

- (V-1) $\ell = \ell'$ and $C = \texttt{true}$, and
- (V-2) For all ℓ, $\Omega, \ell \vdash v : \tau, \ell, C$.

Lemma (Canonical Forms): If $\Sigma, \ell, \vdash v : t\langle \ell_v \rangle, \ell', C$, then

- If $t = \texttt{addr}(T)$ then ℓ_v is a concrete location z, and $v = \texttt{addr}(z)$, and $\Sigma.\mathbb{G}[z] = T$.
- If $t = \texttt{Bool}$ then either $v = \texttt{true}$ or $v = \texttt{false}$.
- If $t = (t_0, t_1)$ then $v = (v_0, v_1)$, and $\Sigma, \ell \vdash v_0 : t_0\langle \ell_v.0 \rangle, \ell', \texttt{true}$ and $\Sigma, \ell \vdash v_1 : t_1\langle \ell_v.1 \rangle, \ell', \texttt{true}$
- If $t = \texttt{vector}(t, k)$ then $k \in \mathbb{N}$, $v = \texttt{vector}(v_0, \ldots, v_{k-1})$, and for all $0 \leq i < k$, $\Sigma, \ell \vdash v_i : t\langle \ell_v.i \rangle, \ell', \texttt{true}$
- If $t = \forall \overline{\kappa}, \overline{\alpha}.C_f \Rightarrow (\tau_{in}, \ell_{in}) \rightarrow (\tau_{out}, \ell_{out})$, then $v = \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}]\ (id : \tau_{in}, \ell_{in}) \rightarrow e$ and $\{\overline{\kappa}\} \cup \{\overline{\alpha}\} \vdash \tau_{in}, \ell_{in}$ and $\mathbb{G}, \{\overline{\kappa}\} \cup \{\overline{\alpha}\}, \{id := \tau_{in}\}, \emptyset, \ell_{in} \vdash e : \tau_{out}, \ell_{out}, C_f$.
- If $t = \texttt{Unit}$ then $v = ()$

A.3.2 Minor Lemmas

Lemma (Rounding Lemma): For all ℓ, b:

- $b \notin \texttt{round}(\ell, b)$, and
- $\ell \leq \texttt{round}(\ell, b)$, and
- for all k, $\ell[k/b] \leq \texttt{round}(\ell, b)$
- if $\Delta, (\mathbb{K}, b \leq k) \vdash \ell$ then $\Delta, \mathbb{K} \vdash \texttt{round}(\ell, b)$

Proof: The fact that $b \notin \texttt{round}(\ell, b)$ is immediate, since it is required for `drop` to terminate. If $b \notin \ell$ then $\ell = \texttt{round}(\ell, b)$. Otherwise, note that `drop` returns a prefix of ℓ, so adding 1 to it results in something strictly larger than ℓ. As a result, we have $\ell[k/b] \leq \texttt{round}(\ell, b)[k/b] = \texttt{round}(\ell, b)$. Finally, since `drop` returns a prefix of ℓ that does not include b, if if $\Delta, (\mathbb{K}, b \leq k) \vdash \ell$ then we know that $\Delta, \mathbb{K} \vdash \texttt{round}(\ell, b)$

Lemma (C.2.1): If $i \leq j$ then for all ℓ, $\ell[i/b] \leq \ell[j/b]$.
Proof: $\ell[i/b]$ and $\ell[j/b]$ are identical in all entries that do not involve b, and in those entries we can prove that $\ell[j/b]$ is no smaller.

A.3.3 Well-formedness lemmas

Lemma (Well-formed outputs): If $\vdash \Omega$ and $\Omega \vdash \ell_{start}$, and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then $\Omega \vdash \tau, \ell_{end}, C$

Rather than prove this lemma directly, we will first prove a slightly stronger lemma (the wellformed-helper lemma), which has useful corollaries. Once that is done, this lemma is immediate by combining it with the additional premise that $\Omega \vdash \ell_{start}$.

Lemma (Wellformed-helper): If $\vdash \Omega$ and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then $\Omega \vdash \tau$ and either $\Omega \vdash \ell_{end}$ or $\ell_{end} = \ell_{start}$. Furthermore, for each constraint $x \leq y$ in C, $\Omega \vdash y$ and either $\Omega \vdash x$ or $x = \ell_{start}$.

Proof: Structural induction on the typing judgement.

Case UNIT: We have $\Omega \vdash \ell'$, so $\Omega \vdash \texttt{Unit}\langle \ell' \rangle$. The rest is immediate.

Case PAIR: By induction, either $\Omega \vdash \ell_1$ or $\ell_1 = \ell_0$. Similarly, either $\Omega \vdash \ell_2$ or $\ell_1 = \ell_0$. We have three possibilities: either (1) $\Omega \vdash \ell_2$ or (2) $\ell_2 = \ell_1 = \ell_0$ or (3) $\ell_2 = \ell_1$ and $\Omega \vdash \ell_1$ (in which case $\Omega \vdash \ell_2$). The fact that $\Omega \vdash t_1\langle \ell.0 \rangle, t_2\langle \ell.1 \rangle$ is immediate by induction.

Finally, by induction we know that our requirements are met for C_1. For C_2, we know by induction that for each constraint $x \leq y \in C_2$, $\Omega \vdash y$, and either $\Omega \vdash x$ or $x = \ell_1$. But in the latter case, we already know that either $\ell_1 = \ell_0$ or $\Omega \vdash \ell_1$, so we are done.

Case LET: We start by using induction on the first premise. This tells us that $\Omega \vdash \tau_1$, so $\vdash \Omega.(\Gamma[id := \tau_1)$ and we can continue using induction on the second premise. The rest is analogous to the PAIR case.

Case DEREF: The proof that $\Omega \vdash \tau$ is analogous to the UNIT case. By induction, $\Omega \vdash \ell_2$, so $\Omega \vdash S(\ell_2)$. Our requirements for C are satisfied by induction, so we need only show that $\Omega \vdash \ell_2$ and either $\Omega \vdash \ell_1$ or $\ell_1 = \ell_0$. Both are immediate by induction.

Case COMP: We may assume by alpha-renaming that $b \notin \mathbb{K}$. Thus since we know that $\Omega \vdash k$, we can safely use induction on our typing premise. After this, the proof that $\Omega \vdash \tau$ is straightforward.

By induction, we know that either $\Delta, (\mathbb{K}, b < k) \vdash \ell_{end}$ or $\ell_{end} = \alpha_{start}$. In the former case, $\ell_{end}[\ell_{init}/\alpha_{start}] = \ell_{end}$, so by the rounding lemma $\Omega \vdash \texttt{round}((, \ell)_{end}, b)$. Otherwise,

$\ell_{end}[\ell_{init}/\alpha_{start}] = \ell_{init}$.

We can use the same logic on the first element of each constraint in C_0, C_1, and C_2. The claim for the second element follows more easily by induction. Finally, note that the `round` function does not add any free variables.

Case ABS: Since we have as a premise that $\Omega \vdash \ell'$ and $\Omega \vdash t_f$, this case is immediate.

Case APP: By induction, we know that $\Omega \vdash t_f$, and therefore that $\Omega.\Delta \cup \overline{\kappa} \cup \overline{\alpha}, \Omega.\mathbb{K} \vdash \tau_{out}, \ell_{in}, \ell_{out}, C_f$. Therefore, since $\Omega \vdash \overline{k}, \overline{\ell}$, we have that $\Omega \vdash \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$ and similarly for ℓ_{in}, ℓ_{out} and C_f. By induction we also know that either $\Omega \vdash \ell_2$ or $\ell_2 = \ell_{start}$. Thus all we need to show is that our desired property holds for C_1 and C_2, which follows the same pattern as the PAIR case.

Lemma (F-1): If $\vdash \Omega$, $\alpha \notin \Omega.\Delta$, and $\Omega, \alpha \vdash e : \tau, \ell_{end}, C$, then either $\ell_{end} = \alpha$ or α does not appear in ℓ_{end}. Furthermore, for each constraint $x \leq y$ in C, α does not appear in y, and if α appears in x then $x = \alpha$.
Proof: Immediate upon application of the wellformed-helper lemma.

A.3.4 Constraint Lemmas

Lemma (Monotonicity): If $\vdash \Omega$ and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then $C \Rightarrow \ell_{start} \leq \ell_{end}$.
Proof: Structural induction on the typing judgement.

Case PAIR: By induction, $C_1 \Rightarrow \ell_0 \leq \ell_1$ and $C_2 \Rightarrow \ell_1 \leq \ell_2$, so by transitivity $C_1 \wedge C_2 \Rightarrow \ell_0 \leq \ell_2$ as required.

Case DEREF: By induction, $C \Rightarrow \ell_0 \leq \ell_1$, so by transitivity $C \wedge \ell_1 \leq \ell_2 \Rightarrow \ell_0 \leq \ell_2$.

Case LOOP: By induction, $C \Rightarrow \alpha_{start} \leq \ell_{end}$. Since substitution preserves implica-

tion, C_0 implies that

$$\ell_{init} = \alpha_{start}[\ell_{init}/\alpha_{start}][0/b] \leq \ell_{end}[\ell_{init}/\alpha_{start}][0/b],$$

and from the rounding lemma we get that

$$\ell_{end}[\ell_{init}/\alpha_{start}][0/b] \leq \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b)$$

Case APP: By the well-formedness lemma, $\Omega \vdash t_f$, so $C_f \Rightarrow \ell_{in} \leq \ell_{out}$. Since substitution preserves implication, $C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \Rightarrow \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \leq \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$. The rest is straightforward by transitivity.

Lemma (Bounded Constraints): If $\vdash \Omega$ and $\Omega, \ell_{start} \vdash e : \tau, \ell_{end}, C$, then for each constraint $x \leq y$ in C we have that $C \Rightarrow \ell_{start} \leq x \leq y \leq \ell_{end}$.
Proof: Induction on the typing judgement.

Case PAIR: By induction on the first premise, we have that for each constraint $x \leq y$ in C_1, $C_1 \Rightarrow \ell_0 \leq x \leq y \leq \ell_1$. By monotonicity on the second premise, $C_2 \Rightarrow \ell_1 \leq \ell_2$. Hence $C_1 \wedge C_2 \Rightarrow \ell_0 \leq x \leq y \leq \ell_2$. The argument for the constraints in C_2 is the analogous, except we use monotonicity to show that $C_1 \Rightarrow \ell_0 \leq \ell_1$.

Case DEREF: By induction, we know that for each constraint $x \leq y \in C$, $C \Rightarrow \ell_0 \leq x \leq y \leq \ell_1$, and $C \wedge \ell_1 \leq \ell_2 \Rightarrow y \leq \ell_1 \leq \ell_2 < S(\ell_2)$. For the final constraint $\ell_1 \leq \ell_2$ note that $\ell_2 < S(\ell_2)$ is immediate, and $C \Rightarrow \ell_0 \leq \ell_1$ follows from monotonicity.

Case LOOP: Let ℓ_{end} be the ending effect of the loop body, and $\ell_{final} = \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b)$ be the ending effect of the original judgement. By induction, we know that for each constraint $x \leq y \in C$, we have $C \Rightarrow \alpha_{start} \leq x \leq y < \ell_{end}$.

By definition of C_0 this means that

$$C_0 \Rightarrow \ell_{init} = \alpha_{start}[\ell_{init}/\alpha_{start}][0/b] \leq x[\ell_{init}/\alpha_{start}][0/b]$$

$$\leq y[\ell_{init}/\alpha_{start}][0/b] < \ell_{end}[\ell_{init}/\alpha_{start}][0/b] = \ell_1 \leq \ell_{final},$$

where the last inequality follows from the rounding lemma. Since we just showed that $C_0 \Rightarrow \ell_{init} \leq \ell_1$. we can use the same logic for C_1, and similarly for C_2.

Case APP: By monotonicity, we quickly conclude that the constraints imply that

$$\ell_0 \leq \ell_1 \leq \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \leq \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}].$$

Thus our property holds for the final constraint $\ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$, and for all constraints in C_1 and C_2 by induction. For constraints in $C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$, our property holds because $\Omega \vdash t_f$, by well-formedness.

A.3.5 Weakening Lemmas

Lemma (Location Weakening): Assume $\vdash \Omega$ and $\Omega, \ell_{start}, \vdash e : \tau, \ell_{end}, C$ where $\vDash C$. Then for all $\ell'_{start} \leq \ell_{start}$, then there is some $\ell'_{end} \leq \ell_{end}$ such that $\Omega, \ell'_{start}, \vdash e : \tau, \ell'_{end}, C'$, where $\vDash C'$. Furthermore, either $\ell'_{end} = \ell_{end}$ or $\ell'_{end} = \ell'_{start}$.

Proof: Structural induction over the typing relation.

Case UNIT: In this case, $\ell'_{end} = \ell'_{start}$ and $\ell'_{start} \leq \ell_{start} = \ell_{end}$, and of course $C' = \texttt{true}$ is valid.

Case PAIR: By induction, $\Omega, \ell'_{start} \vdash e_1 : \tau_1, \ell'_1, C'_1$ where $\vDash C'_1$, $\ell'_1 \leq \ell_1$, and either $\ell'_1 = \ell'_{start}$ or $\ell'_1 = \ell_1$. In the latter case, we may re-use our second premise and we are done. Otherwise, we can use induction on the second premise to get that

$\Omega, \ell'_{start} \vdash e_2 : \tau_2, \ell'_2, C'_2$, where $\vDash C'_2$, $\ell'_2 \leq \ell_2$, and either $\ell'_2 = \ell_2$ or $\ell'_2 = \ell'_{start}$. We can then reapply the PAIR rule to finish the case.

Case DEREF: By induction, we can show that $\Omega, \ell'_{start} \vdash e : \texttt{addr}(T)\langle \ell_2 \rangle, \ell'_1, C'$ where $\ell'_1 \leq \ell_1$ and $\vDash C'$. Since our original constraints were valid, we know that $\vDash \ell_1 \leq \ell_2$, so by transitivity $\vDash \ell'_1 \leq \ell_2$. We can thus apply the DEREF rule, noting that $\ell'_{end} = S(\ell_2) = \ell_{end}$.

Case IF-LEFT: By induction, we can show that $\Omega, \ell'_{start} \vdash e_1 : \texttt{Bool}\langle \ell \rangle, \ell'_1, C'_1$, where $\vDash C'_1$ and either $\ell'_1 = \ell_1$ or $\ell'_{start} = \ell'_1 \leq \ell_1$. In the former case, we may re-use the other two premises and finish immediately. Otherwise, by induction $\Omega, \ell'_{start} \vdash e_2 : \tau, \ell'_2, C'_2$ and $\Omega, \ell'_{start} \vdash e_3 : \tau, \ell'_3, C'_3$, where $\vDash C'_2 \wedge C'_3$, and ℓ'_2 and ℓ'_3 are each either equal to ℓ_{start} or ℓ_2 and ℓ_3, respectively.

We have as a premise that $\ell_2 \leq \ell_3$. Thus if $\ell'_3 = \ell_3$, then by transitivity $\ell'_2 \leq \ell_2 \leq \ell_3 \leq \ell'_3$, and we may reapply the IF-LEFT rule. Otherwise, $\ell'_3 = \ell'_{start}$, and since $\vDash C_2$ we may apply monotonicity to learn that $\ell'_{start} \leq \ell'_2$, so we may apply the IF-RIGHT rule.

Case LOOP: In this case, let ℓ_{end} refer to the output of the loop body typing judgement, and let ℓ_{final} refer to the output of the original typing judgement. We do not need induction here; we can simply reapply the LOOP rule with $\ell_{start'}$ instead of ℓ_{start}. We still need to show that our output satisfies the desired properties, however.

By applying Lemma F-1 to our typing premise, we get that either $\ell_{end} = \alpha_{start}$ or ℓ_{end} does not contain α_{start}. In the former case, $\ell_{end}[\ell'_{start}/\alpha_{start}] = \ell'_{start}$. By alpha-renaming, we may assume that b does not appear in ℓ'_{start}; thus $\texttt{round}(\ell'_{start}, b) = \ell'_{start}$. If ℓ_{end} does not contain α_{start}, then $\texttt{round}(\ell_{end}[\ell'_{start}/\alpha_{start}], b) = \texttt{round}(\ell_{end}[\ell_{start}/\alpha_{start}], b) = \ell_{final}$.

Also by lemma F-1, we get that for each constraint $x \leq y$ in C, α_{start} appears only in x, and only if $x = \alpha_{start}$. Thus for any ℓ, ℓ' with $\ell \leq \ell'$, $x[\ell/\alpha_{start}] \leq x[\ell'/\alpha_{start}] \leq$

$y[\ell'/\alpha_{start}] = y[\ell/\alpha_{start}] = y$. If b does not appear in ℓ or ℓ', then the substitutions for α_{start} will commute with those for b, so if we know that $x[\ell'/\alpha_{start}][i/b] \leq y[\ell'/\alpha_{start}][i/b]$ we will also have $x[\ell/\alpha_{start}][i/b] \leq y[\ell/\alpha_{start}][i/b]$.

We may assume that b does not appear in ℓ_{start} or ℓ'_{start} by alpha-renaming. Thus we know that our new version of $\vDash C_0$ immediately. Similarly, we know that our new ℓ_1 does not contain b, and is at most the original ℓ_1, so we may apply the same logic to show that our new C_1 is valid. Finally, we repeat the process to show that our new C_2 is valid.

Case APP: As in the PAIR case, we use induction to show that $\Omega, \ell'_{start} \vdash e_1 : \tau_f, \ell'_1, C'_1$, and if $\ell'_1 = \ell'_{start}$ we continue to show that $\Omega, \ell'_{start} \vdash e_2 : \tau_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell'_2, C'_2$, where $\vDash C'_1 \wedge C'_2$. Because $\vDash C$, we note that $\ell'_2 \leq \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$, so we may reapply the APP rule with the same ℓ_{end} and C_f as before.

Definition: If we have two maps M, M', we say that $M' \supseteq M$ if $M[k] = v$ implies $M'[k] = v$. Say that $\Omega' \supseteq \Omega$ if $\Omega'.\mathbb{G} \supseteq \Omega.\mathbb{G}$ and $\Omega'.\Delta \supseteq \Omega.\Delta$ and $\Omega'.\Gamma \supseteq \Omega.\Gamma$ and $\Omega'.\mathbb{K} \supseteq \Omega.\mathbb{K}$.

Lemma (Environment Weakening): If $\vdash \Omega$ and $\Omega, \ell_{start} \vdash e : \tau, \ell_{start}, C$, then for all $\Omega' \supseteq \Omega$ where $\vdash \Omega'$ we have $\Omega', \ell_{start} \vdash e : \tau, \ell_{end}, C$.

Proof: Straightforward structural induction over the typing derivation.

A.3.6 Substitution Lemmas

Lemma (Substitution Lemma for ids): If $\Omega, \ell \vdash v : \tau_1, \ell, \texttt{true}$ and $\Omega.(\Gamma[id := \tau_1]), \ell \vdash e : \tau_2, \ell', C$, then $\Omega, \ell \vdash e[v/id] : \tau_2, \ell', C$.

Proof: Standard induction over the typing derivation.

Lemma (Substitution Lemma for bs): If $n < n'$, and $\Omega.(\mathbb{K}, b < n'), \ell \vdash e : \tau, \ell', C$, then we have $\Omega[n/b], \ell[n/b] \vdash e[n/b] : \tau_2[n/b], \ell'[n/b], C[n/b]$.

Proof: Structural induction over the typing derivation. Mostly straightforward.

Case INDEX-VAR: If this b is not our target b, then the claim follows by straightforward induction. Otherwise, since we assumed that $n < n'$, we will be able to use the INDEX-CONST rule after substituting.

Case COMP: We may assume by alpha-renaming that the loop's b is not our target b. Since we are substituting in a natural number, we cannot create any repeated indices, so the rest of the claim follows by induction.

Lemma (Susbtitution Lemma for αs): If $\Delta' = \Delta \cup \{\alpha\}$ and $\Delta, \mathbb{K} \vdash \ell_\alpha$ and $\mathbb{G}, \Delta', \mathbb{K}, \Gamma, \ell \vdash e : \tau, \ell', C$, then $\mathbb{G}, \Delta, \Gamma[\ell_\alpha/\alpha], \mathbb{K}, \ell[\ell_\alpha/\alpha] \vdash e[\ell_\alpha/\alpha] : \tau[\ell_\alpha/\alpha], \ell'[\ell_\alpha/\alpha], C[\ell_\alpha/\alpha]$

Proof: Structural induction over the typing derivation.

Case UNIT: We have the assumption that $\Delta, \mathbb{K} \vdash \ell_\alpha$ and the premise that $\Delta', \mathbb{K} \vdash \ell'$, so $\Delta \vdash \ell'[\ell_\alpha/\alpha]$ and we may reapply the UNIT rule.

Case LOOP: By alpha-renaming, we may assume that b does not appear in our current environment. Thus since $\mathbb{K} \vdash \ell_\alpha$, ℓ_α does not contain any instances of b, so substituting it into a constraint will not add any instances. Thus since we have the premise $\texttt{nri}(C, b)$, we conclude that $\texttt{nri}(C, [\ell_\alpha/\alpha])$.

Since $\Delta' \not\vdash \alpha_{start}$, we have that $\alpha_{start}[\ell_\alpha/\alpha] = \alpha_{start}$, so our typing premise still has the right form after induction. The rest is straightforward.

Case ABS: The only nontrivial thing we need to show is that after substituting into e, our output type is $t_f[\ell_\alpha/\alpha]$. By alpha-renaming, we may assume that our α does not appear in $\overline{\alpha}$, and the rest is straightforward.

Case APP: We again use alpha-renaming to assume that our α does not appear in the $\overline{\alpha}$ inside τ_f, nor does any element of Δ. Thus since $\Omega \vdash \ell_\alpha$, substitutions for ℓ_α

commute with substitutions for $\overline{\alpha}$. The rest is straightforward.

Lemma (Substitution Lemma for κs): If $\Delta' = \Delta \cup \{\kappa\}$, and $\Delta, \mathbb{K} \vdash k_\kappa$, and $\mathbb{G}, \Delta', \mathbb{K}, \Gamma, \ell \vdash e : \tau, \ell', C$, then $\mathbb{G}, \Delta, \Gamma[k_\kappa/\kappa], \mathbb{K}[k_\kappa/\kappa], \ell[k_\kappa/\kappa] \vdash e[k_\kappa/\kappa] : \tau[k_\kappa/\kappa], \ell'[k_\kappa/\kappa], C[k_\kappa/\kappa]$

Proof: Structural induction over the typing derivation.

Case UNIT: Identical to the UNIT case from the previous proof.

Case INDEX-VAR: When we use induction on our premise, we get that the new output type is $\texttt{vectort}[k_\kappa/\kappa], k[k_\kappa/\kappa]$, and $(\Omega[k_\kappa/\kappa]).\mathbb{K}[b] = k[k_\kappa/\kappa]$, so we may reapply the rule.

Case LOOP/ABS/APP: Analogous to the cases from the previous lemma.

A.3.7 Loop lemmas

Lemma (Loop Unrolling Helper): Let x and y be effects, each of which contains at most one variable i, which is a b. Assume that that $x[0/i] \leq y[0/i] \leq x[1/i] \leq y[1/i] \leq x[2/i] \leq y[2/i]$. Then, taking the list-based view of effects, one of the following is true for each index j:

- $x_j = y_j$, or
- there is some previous index $j' < j$ where $x_{j'} = y_{j'} = i + n$ for some $n \in \mathbb{N}$.

Proof: Assume towards a contradiction there's some index that doesn't satisfy either of these; let j be the first such index. Since j is the first, x and y are identical at each previous index, and since j fails the second point x and y must be constants at each prior index. Thus we cannot have $x_j < y_j$ or $x_j > y_j$, since this would fail one of the inequalities $x[0/i] \leq y[0/i] \leq x[1/i]$. So x_j and y_j must be incomparable; the only way for this to happen is for one to be a constant m and the other to be $i + n$, where $m, n \in \mathbb{N}$.

We now have four cases:

1. If $x_j = i + n$ and $y_j = m$ with $n \geq m - 1$ then we would have $x[2/i] > y[2/i]$, a contradiction.

2. If $x_j = i + n$ and $y_j = m$ with $n < m - 1$ then we would have $x[1/i] < y[0/i]$, a contradiction.

3. If $x_j = m$ and $y_j = i + n$ with $n \geq m$ then we would have $y[1/i] > x[2/i]$, a contradiction.

4. If $x_j = m$ and $y_j = i + n$ with $n < m$ then $y[0/i] < x[0/i]$, a contradiction.

Lemma (Loop Unrolling): Assume $\vdash \Omega$ and $\Omega, \alpha_{start} \vdash e : \tau, \ell_{end}, C$. For all ℓ_{init} and bounded sizes i, define $\ell_0 = \ell_{init}$, $C_0 = C[\ell_0/\alpha_{start}][0/i]$ and for $j > 0$ define $\ell_j = \ell_{end}[\ell_{j-1}/\alpha_{start}][(j-1)/i]$ and $C_j = C[\ell_j/\alpha_{start}][j/i]$. Finally, assume $\texttt{nri}(C, i)$. Then if M is a model of $C_0 \wedge C_1 \wedge C_2$, M is also a model of $\forall j \geq 0.C_j$.
Proof: Let M be a model of $C_0 \wedge C_1 \wedge C_2$, and replace all variables in C with their values in M. After doing so, the only variables in C are α_{start} and i.

Pick a constraint $x \leq y$ in C (if none exist, the lemma is trivial). Note that by lemma F-1, α_{start} doesn't appear in y, so we may ignore α_{start} substitutions into it.Because $C_0 \wedge C_1 \wedge C_2$ is true in this model, by the bounded constraints lemma we get

$$\ell_0 \leq x[\ell_0/\alpha_{start}][0/i] \leq y[0/i] \leq \ell_{end}[\ell_0/\alpha_{start}][0/i] = \ell_1$$

$$\ell_1 \leq x[\ell_1/\alpha_{start}][1/i] \leq y[1/i] \leq \ell_{end}[\ell_1/\alpha_{start}][1/i] = \ell_2$$

$$\ell_2 \leq x[\ell_2/\alpha_{start}][2/i] \leq y[2/i] \leq \ell_{end}[\ell_2/\alpha_{start}][2/i] = \ell_3$$

Again by Lemma F-1, if α_{start} appears in ℓ_{end} then $\ell_{end} = \alpha_{start}$. In this case,

$\ell_0 = \ell_1 = \ell_2 = \ell_3$, so our chain of inequalities above is in fact a chain of equalities. Thus since y and $x[\ell_0/\alpha_{start}]$ don't change when we substitute i into them, they must both be constants (since they contain no other variables), and so the constraint holds regardless of i.

Otherwise, ℓ_{end} does not contain α_{start}, so we may ignore that substitution. We now split into cases, based on whether or not $x = \alpha_{start}$.

If so, then we can consolidate our above inequality chains into

$$y[0/i] \leq \ell_{end}[0/i] \leq y[1/i] \leq \ell_{end}[1/i] \leq y[2/i] \leq \ell_{end}[2/i].$$

Since we have substituted every variable except i, we can use our helper lemma to show that there is some index j such that $\ell_{end_j} = y_j = i + n$, and for all prior indices ℓ_{end} and y are identical constants. Thus for all $j > 0$,

$$x[\ell_j/\alpha_{start}][j/i] = \ell_j[j/i] = \ell_{end}[(j-1)/i] < y[j/i] = y[\ell_j/\alpha_{start}][j/i].$$

Thus the constraint $x \leq y$ holds in all C_j for $j > 0$.

If $x \neq \alpha_{start}$, by lemma F-1 α_{start} does not appear in x, so we may consolidate our big inequality into

$$x[0/i] \leq y[0/i] \leq x[1/i] \leq y[1/i] \leq x[2/i] \leq y[2/i].$$

As in the previous case, we apply our helper lemma to conclude that x and y are identical up to some index j, where $x_j = y_j = i + n$ for some $n \in \mathbb{N}$. Since we have as a premise that $\texttt{nri}(C, i)$, neither x nor y contain multiple copies of i. Since i was the only variable remaining, this means that all future entries are constants. Thus the proof that $x[0/i] \leq y[0/i]$ shows that $x[j/i] \leq y[j/i]$ for all $j \geq 0$, and so the constraint holds in all C_j.

Since $x \leq y$ was an arbitrary constraint in C, this argument works for each constraint individually, so by combining them we have shown that each constraint in C_j is true for all $j > 0$; since we already know that C_0 is satisfied, we have shown $\forall j \geq 0.C_j$ as required.

A.4 Proof of Soundness

Theorem (Soundness): Let $\Sigma, z \vdash e : \tau, z'', C$ where $\vdash \Sigma$ and $\vDash C$. Then either e is a value or there are some M', z', e' such that $M, z, e \to M', z', e'$. Furthermore, $M' \sim \Sigma.\mathbb{G}$, and $\Sigma, z' \vdash e' : \tau, z'', C'$ where $\vDash C'$.

As usual, we prove this theorem in two parts: progress and preservation.

Theorem (Progress):: Let $\Sigma, z \vdash e : \tau, z', C$ where $\vDash C$. Let $M \sim \Sigma.\mathbb{G}$. Then either e is a value or there are some M', z'', e' such that $M, z, e \to M', z'', e'$.

Proof: Structural induction on the typing derivation.

Case UNIT/TRUE/FALSE/ADDR/ABS: In these cases e must be a value, so we are done.

Case VAR: Since $\Sigma.\Gamma = \emptyset$, this case is impossible.

Case PAIR: Here, $e = (e_1, e_2)$ and $C = C_1 \wedge C_2$. Since $\vDash C$, $\vDash C_1$; thus by induction, either e_1 is a value or there is some M', z'', e_1' such that $M, z, e_1 \to M', z'', e_1'$. In the former case, we may apply the PAIR-2 rule; in the latter case, we may apply the PAIR-1 Rule.

Case FST: Here $e = \texttt{fst}\ e_1$. By induction, either e_1 is a value or it steps to something. In the latter case, we may apply FST-1. Otherwise, by canonical forms $e_1 = (v_0, v_1)$, so we may apply FST-2.

Case SND: Analogous to the FST case.

Case VECTOR: Here, $e = \texttt{vector}(v_0, \ldots, v_n, e_0, \ldots, e_m)$. If all entries of e are values, then e is a value and we are done. Otherwise, by induction e_0 steps to something, and thus we may apply the VECTOR rule.

Case LET: Here, $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$. If e_1 is a value then we may apply LET-2, otherwise by induction we may apply LET-1.

Case DEREF: Here, $e = !e_1$. If e_1 is not a value, then by induction we may apply DEREF-1. Otherwise, by canonical forms, ℓ_2 is a concrete effect z_e and $e_1 = \texttt{addr}(z_e)$. Furthermore, since e_1 is a value, $\ell_1 = \ell_0 = z$. Since $\vDash C$, $z = \ell_1 \leq \ell_2 = z_e$. Also by canonical forms, $\mathbb{G}[z_e] = T$. Thus, since $M \sim \mathbb{G}$, $M[z_e]$ exists. This is sufficient to apply the DEREF-2 rule.

Case UPDATE: Here, $e = e_1 := e_2$. By induction, either e_1 is a value or it steps, and similarly for e_2. If e_1 steps then we may apply UPDATE-1; otherwise, if e_2 steps we may apply UPDATE-2. If both are values, then $z = \ell_0 = \ell_1 = \ell_2$. By canonical forms, ℓ_3 is a concrete effect z_e, $e_1 = \texttt{addr}(z_e)$, and $\mathbb{G}[z_e] = T$. Since $\vDash C$, $z = \ell_2 \leq \ell_3 = z_e$, and since $M \sim \mathbb{G}$, $M[z_e]$ exists. Thus we may apply the UPDATE-3 rule.

Case IF-LEFT: Here $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. By induction, either e_1 is a value or it steps. In the latter case, we may apply IF-1. In the former case, by canonical forms e_1 is either `true` or `false`, so we may apply either IF-TRUE or IF-FALSE accordingly.

Case IF-RIGHT: Identical to IF-LEFT.

Case INDEX-CONST: Here $e = e_1[n]$. By induction, either e_1 is a value or it steps. In the latter case, we may apply INDEX-1. Otherwise, by canonical forms $e_1 = \texttt{vector}(v_0, \ldots, v_{n'-1})$. We have the premise that $n < n'$, so $n \leq n' - 1 = m$ and we may apply the INDEX-2 rule.

Case INDEX-VAR: Since $\Sigma.\mathbb{K}$ is empty, this case cannot occur.

Case LOOP: Since $\Sigma \vdash k$ and $\Sigma.\Delta, \Sigma.\mathbb{K}$ are empty, k must not be a polymorphic variable or b. Hence $k \in \mathbb{N}$, so we may apply the LOOP rule.

Case COMP: Identical to LOOP.

Case APP: Here $e = e_0\ [\overline{k}, \overline{\ell}]\ e_1$. By induction, e_0 either steps or is a value, and similarly for e_1. If e_1 steps, we may apply APP-1; otherwise, if e_2 steps, we may apply APP-2. If both are values, then by canonical forms v is a function value, so we may apply APP-3.

Theorem (Preservation): Let $\Sigma, z_{start} \vdash e : \tau, z_{end}, C$ and $M, z_{start}, e, \rightarrow M', z_{step}, e'$, where $\vDash C$ and $M \sim \Sigma.\mathbb{G}$. Then $M' \sim \Sigma.\mathbb{G}$, and $\Sigma, z_{step} \vdash e' : \tau, z'_{end}, C'$, where $\vDash C'$ and $z'_{end} \leq z_{end}$.

Proof: Structural induction on the typing derivation.

Case UNIT/TRUE/FALSE/ADDR/ABS: No operational semantics rule applies to these expressions (since they are values), so these cases are impossible.

Case VAR: This case is impossible because $\Sigma.\Gamma$ is empty.

Case PAIR: In this case, $e = (e_1, e_2)$. We have two typing premises, $\Sigma, z \vdash e_1 : \tau_1, \ell_1, C_1$ and $\Sigma, \ell_1 \vdash e_2 : \tau_2, z_{end}, C_2$. Since $\vDash C_1 \wedge C_2$, $\vDash C_1$ and $\vDash C_2$. We now have two options for which operational semantics rule we used.

If we used the PAIR-1 rule, then $M, z_{start}, e_1 \rightarrow M', z_{step}, e'_1$ and $e' = (e'_1, e_2)$. By induction on the first typing premise, we conclude that M' is well-formed, and $\Sigma, z_{step} \vdash e'_1 : \tau_1, \ell'_1, C'_1$, where $\vDash C'_1$ and $\ell'_1 \leq \ell_1$.

We may then apply weakening to the second premise, to get that $\Sigma, \ell'_1 \vdash e_2 : \tau_2, z'_{end}, C'_2$, where $\vDash C'_2$ and $z'_{end} \leq z_{end}$. Combining these two premises is enough to show that $\Sigma, z_{step} \vdash e' : \tau, z'_{end}, C'_1 \wedge C'_2$; note that $\vDash C'_1 \wedge C'_2$ since both conjuncts are valid.

If we used the PAIR-2 rule, then e_1 is a value, so by applying lemma V-1 to the first typing premise, we get that $\ell_1 = z_{start} = z_{step}$. We may then use induction on the second premise to find that M' is well-formed and $\Sigma, z_{step} \vdash e_2' : \tau_2, \ell_2', C_2'$, where $\vDash C_2'$ and $\ell_2' \leq \ell_2$. We can then combine this with the first typing premise to apply the PAIR typing rule.

Case FST: In this case, $e = \texttt{fst}\ e_1$. We have two options for which operations semantics rule we used. The case for the FST-1 rule is analogous to the case for the PAIR-1 rule. If we used the FST-2 rule, then $M' = M$ is well-formed, $z_{start} = z_{step}$, and $e_1 = (v_1, v_2)$ and $e' = v_1$. By canonical forms, we know that $\Omega, z_{start} \vdash v_1 : t_1\langle \ell_v.1 \rangle, z_{end}, \texttt{true}$, which is exactly what we needed to show.

Case SND: Analogous to case FST.

Case VECTOR: In this case, e is a vector expression, so we must have used the VECTOR operational semantics rule. Hence $e = \texttt{vector}(v_0, \ldots, v_n, e_0, \ldots, e_m)$, where e_0 is the first non-value subexpression. Furthermore, $e' = \texttt{vector}(v_0, \ldots, v_n, e_0', \ldots, e_m)$.

From the premises of the typing judgement, we know that $\Sigma, z_{start} \vdash v_0 : \tau_1, \ell_1, C_1$, $\Sigma, \ell_1 \vdash v_1 : \tau_2, \ell_2, C_2$, and so forth until $\Sigma, \ell_n \vdash e_0 : \tau_{n+1}, \ell_{n+1}, C_{n+1}$. Since $\vDash C_1 \wedge C_2 \wedge ... \wedge C_{n+m}$, we have $\vDash C_i$ for each i. Furthermore, since $v_0, \ldots, v_n$ are values, by lemma V-1 we have $z_{start} = \ell_1 = \cdots = \ell_n$.

Thus by induction, we may show that M' is well-formed and $\Sigma, z_{start} \vdash e_0' : \tau_{n+1}, z_{n+1}', C_{n+1}'$. where $\vDash C_{n+1}'$ and $z_{n+1}' \leq z_{n+1}$. By lemma V-2, we may replace the location z_{start} with z_{step} in all of the value typing judgements, and by weakening we may change the starting location of the e_1 judgement (if it exists) to z_{n+1}'. By combining our new judgment with the replaced judgements and with the remainder of the original typing premises, we can prove that $\Sigma, z_{step} \vdash e' : \tau, z_{end}', C_1 \wedge C_2 \wedge \cdots \wedge C_{n+1}' \wedge C_{n+2}' \wedge \cdots \wedge C_{n+m}$; note that the output constraints are valid because each of the components is valid. Furthermore, either $z_{end}' = z_{end}$ (if e_0 was not the last

component), or $z_{end'} = z'_{n+1} \leq z_{n+1} = z_{end}$ (if e_0 was the last component). In either case, $z'_{end} \leq z_{end}$ as required.

Case LET: We have two cases for which operational semantics rule we used. The LET-1 case is analogous to PAIR-1. If we used LET-2, then $M' = M$ is well-formed, $z_{step} = z_{start}$, $e = \texttt{let}\ x = v\ \texttt{in}\ e_1$, and $e' = e_1[v/x]$. By applying lemma V-2 to the first typing premise, we obtain that $\Sigma, z_{start} \vdash v : \tau_1, z_{start}, \texttt{true}$ and $\Sigma.(\Gamma[x := \tau_1]), z_{start} \vdash e : \tau, z_{end}, C_2$. By the substitution lemma for xs, we may turn the latter premise into $\Sigma, z_{start} \vdash e[v/x] : \tau, z_{end}, C_2$. Since $\vDash C_2$, this is exactly what we needed to show.

Case DEREF: As before, we have two cases for the operational semantics rule, and the DEREF-1 case is analogous to the PAIR-1 case. If we used DEREF-2, then $e = !\texttt{addr}(z_e)$, $M' = M$ is well-formed, $z_{start} \leq z_e$, and $z_{step} = S(z_e)$. Furthermore, our typing premise becomes $\Sigma, z_{start} \vdash \texttt{addr}(z_e) : \texttt{addr}(T)\langle \ell_2 \rangle, \ell_1, C_1$, and we learn that $\tau = T\langle \ell' \rangle$ and $z_{end} = \ell_2$. Finally, since $\vDash C_1 \wedge \ell_1 \leq \ell_2$, we know that $\ell_1 \leq \ell_2$.

By applying canonical forms to our typing premise, we get that $\ell_2 = z_e$ and $\Sigma.\mathbb{G}[z_e] = T$. Since M is well-formed, $\Sigma, S(z_e) \vdash M[z_e] : T\langle \ell' \rangle, S(z_e), \texttt{true}$, which is what we needed to show.

Case UPDATE: Here we have three possible operational semantics rules. UPDATE-1 is anaologous to PAIR-1. UPDATE-2 is similar to UPDATE-1, but we need to use lemma V-2 on the first premise as in the PAIR case.

If we used UPDATE-3, then we get that $e = \texttt{addr}(z_e) := v$, $e' = ()$, $\tau = \texttt{Unit}\langle \ell' \rangle$, and $z_{step} = S(z_e)$. From canonical forms we get that $\Sigma.\mathbb{G}[z_e] = T$ and $\ell_3 = z_e$. By lemma V-1, our second typing premise is now $\Sigma, z_{start} \vdash v : T\langle \ell \rangle, z_{start}, \texttt{true}$, which shows that $M[z_e := v]$ is well-formed. Finally, note that $\ell_{end} = S(\ell_3) = S(z_e) = z_{step}$, and we may immediately show that $\Sigma, S(z_e) \vdash () : \tau, S(z_e), \texttt{true}$ using the UNIT rule.

Case IF-LEFT: We have three cases for the operational semantics rule. The IF-

1 case is analogous to the PAIR-1 case. In both the other cases, we have $e = \texttt{if } v \texttt{ then } e_2 \texttt{ else } e_3$, $M' = M$ is well-formed, and $z_{step} = z_{start}$. Since v is a value, we may apply lemma V-1 to our first typing premise, turning the other judgements into $\Sigma, z_{start} \vdash e_2 : \tau, \ell_2, C_2$ and $\Sigma, z_{start} \vdash e_3 : \tau, \ell_3, C_3$. Finally, we have that $\ell_2 \leq \ell_3 = z_{end}$.

In the IF-TRUE case, we use induction on the first of these premises to get that $\Sigma, z_{step} \vdash e_2 : \tau, \ell_2', C_2'$, where $\vDash C_2'$, and $\ell_2' \leq \ell_2$. Since $\ell_2 \leq \ell_3 = z_{end}$, this is what we needed to show. The IF-FALSE case is similar.

Case IF-RIGHT: Similar to IF-LEFT.

Case INDEX-CONST: Analogous to the FST case

Case INDEX-VAR: Since $\Sigma.\mathbb{K}$ is empty, this case is impossible.

Case LOOP: In this case, we must have used the LOOP operational semantics rule. We know that $e = \texttt{for } b < k \texttt{ do } e_1$, that $k \in \mathbb{N}$, and that $\Sigma.(\mathbb{K}, b < k), \alpha_{start} \vdash e_1 : \tau, \ell_{end}, C_{loop}$. We also know that C_0, C_1 and C_2 are valid. Finally, we know that $M' = M$ is well-formed, that $z_{step} = z_{start}$, and that $e' = e[0/b]; e[1/b]; \ldots; e[k-1/b]; ()$. Recall that this is syntactic sugar for $\texttt{let } x_1 = e[0/b] \texttt{ in let } x_2 = e[1/b] \texttt{ in } \ldots$, where the x_i do not appear anywhere else in the program.

First, we use environment weakening to turn our premise into $\mathbb{G}, \{\alpha_{start}\}, \{b := k\}, \emptyset, \alpha_{start} \vdash e_1 : \tau, \ell_{end}, C_{loop}$. This lets us use the substitution lemmas for αs and bs to show that for all ℓ_α, k' such that $\Sigma \vdash \ell_\alpha$ and $k' < k$ we may turn our typing premise into

$$\Sigma, \alpha_{start}[\ell_\alpha/\alpha_{start}][k'/b] \vdash e_1[\ell_\alpha/\alpha_{start}][k'/b] :$$

$$\tau[\ell_\alpha/\alpha_{start}][k'/b], \ell_{end}[\ell_\alpha/\alpha_{start}][k'/b], C_{loop}[\ell_\alpha/\alpha_{start}][k'/b]$$

Fortunately, we may simplify: τ doesn't matter here, so we drop the substitutions

into it; similarly, since $\alpha_{start} \notin \Sigma$, we may assume by alpha-renaming that α_{start} does not appear in e_1. Thus we end up with

$$\Sigma, \ell_\alpha[k'/b] \vdash e_1[k'/b] : \tau, \ell_{end}[\ell_\alpha/\alpha_{start}][k'/b], C_{loop}[\ell_\alpha/\alpha_{start}][k'/b]$$

Now, by setting $\ell_\alpha = z_{step}$ and $k' = 0$ in our above judgement, we can immediately show that

$$\Sigma, z_{step} \vdash e_1[0/b] : \tau, \ell_{end}[z_{step}/\alpha_{start}][0/b], C_{loop}[z_{step}/\alpha_{start}][0/b].$$

Now define $\ell_0 = z_{start} = z_{step}$ and $C_0 = C[\ell_0/\alpha_{start}][0/b]$, and for $j > 0$ define $\ell_j = \ell_{end}[z_{start}/\alpha_{start}][(j-1)/b]$ and $C_j = C[\ell_j/\alpha_{start}][j/b]$. Note that these definitions are consistent with the ones in the LOOP typing rule. Using these definitions, we can rewrite the above judgement as

$$\Sigma, \ell_0 \vdash e_1[0/b] : \tau, \ell_1, C_0$$

We claim that for all $j > 0$, $\ell_j = \ell_{end}[\ell_{j-1}/\alpha_{start}][(j-1)/b]$. By lemma F-1, either $\ell_{end} = \alpha_{start}$ or α_{start} does not appear in ℓ_{end}. In the former case, $\ell_j = \ell_{end}[z_{start}/\alpha_{start}][(j-1)/b] = z_{start}[(j-1)/b] = z_{start}$. In the latter case, the α_{start} substitution into ℓ_{end} has no effect, so the claim is trivial.

Thus we can apply this same process to show that for $0 \leq j < k$,

$$\Sigma, \ell_j \vdash e_1[0/b] : \tau, \ell_{j+1}, C_j.$$

We can then apply environment weakening again to change Σ into $\Sigma.(\Gamma[x_i = \tau])$ in the above judgement. This allows us to apply the LET rule several times, terminating with the UNIT rule, to obtain the typing judgement we need. There remain two things to show. The first is that $\ell_k \leq \texttt{round}(\ell_{end}[\ell_{init}/\alpha_{start}], b)$, which follows immediately

by definition of ℓ_k and the rounding lemma.

The second thing we must show is that each of the C_j output above is valid. Fortunately, we have shown our definition of ℓ_j and C_j is the same as that in the Loop Unrolling lemma. Thus we may apply the lemma to show that any model for $C_0 \wedge C_1 \wedge C_2$ is also a model for $\forall j \geq 0.C_j$. Since $C_0 \wedge C_1 \wedge C_2$ is valid, this means that each C_j is valid, and we are done.

Case COMP: This is analogous to the LOOP case, except that instead of using the LET rule many times, we apply the VECTOR rule once, again relying on the Loop Unrolling lemma to ensure all the constraints are satisfied.

Case APP: The APP-1 case is analogous to the PAIR-1 case, and the APP-2 case is analogous to the UPDATE-2 case, except that we use the transitivity of $\leq$ to satisfy the last constraint after using effect weakening.

In the APP-3 case, we have that $M' = M$ is well-formed, $z_{step} = z_{start} = \ell_1 = \ell_2$, and $e = v_1\ [\overline{k}, \overline{\ell}]\ v_2$. By canonical forms $v_1 = \texttt{fun}\ [\overline{\kappa}, \overline{\alpha}](x : \tau_{in}, \ell_{in}) \to e_{body}$, where $\Sigma.\mathbb{G}, \{\overline{\kappa}\} \cup \{\overline{\alpha}\}, \emptyset, \{x := \tau_{in}\}, \ell_{in} \vdash e_{body} : \tau_{out}, \ell_{out}, C_f$.

We can then use our substitution lemmas for αs, κs, and xs to turn this typing judgement into

$$\Sigma, \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] \vdash e_{body}[v_2/x][\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] : \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$$

Since we know that $z_{step} = \ell_2 \leq \ell_{in}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$, and since $\vDash C_f[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$, we can apply effect weakening to conclude that there is some $\ell'_{out} \leq \ell_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}]$ and some valid C'_f such that

$$\Sigma, z_{step} \vdash e_{body}[v_2/x][\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}] : \tau_{out}[\overline{\ell}/\overline{\alpha}][\overline{k}/\overline{\kappa}], \ell'_{out}, C'_f$$

This is exactly what we needed to show.

Appendix B

Parasol

B.1 Comparison to hand-optimized code

We also compared Parasol solutions to hand-optimized solutions for three of our applications: Fridge, Conquest, and Starflow. Parasol's solutions performed reasonably close to the hand-optimized solutions for all 3 applications. We describe each application in more detail below.

Fridge (Unbiased RTT) The Fridge[89] data structure is used to collect RTT samples in the data plane by storing requests and matching them with the corresponding response, without penalizing samples with a large RTT. Each request is added to the data structure with probability p, and once a request is in the structure, it can be removed either upon receipt of the response, or if a new response overwrites it when the structure is full.

The value of p is the primary parameter to be optimized. If p is too small, requests are less likely to be added to the structure, and the program will not produce enough RTT samples. Conversely, if p is too large, requests are more likely to be overwritten before their responses arrive.

In general, the objective function that Fridge seeks to minimize is the difference

between the distribution of sampled RTTs and the distribution of all RTTs. We implemented the same error function in Parasol as was used in the original evaluation of Fridge [89]: maximum percentile error, or the maximum error of the sampled distribution for percentiles $\in [5\%, 95\%]$.

In the hand-tuned program, the authors achieved an error of 25%, and our optimized program, found using exhaustive search, achieved a maximum delay estimation error of 18%, well within the expected performance. The Fridge authors found that they could achieve nearly the same error with a wide range of p values. In our workloads, Parasol also found that p had a negligible effect on error as long as it greater than 2^{-10} (0.001) or less than 2^{-3} (0.13). Going outside of that bound for the chosen fridge size increased the error to over 45%.

Conquest. Conquest [23] aims to identify flows that are making a significant contribution to queue build-up, during some time window T. It maintains several sketches as "snapshots" of the queue length for T. During a time window, the program cleans one sketch, writes to one sketch, and reads a flow's queue length estimates from the rest.

Conquest has three parameters that can impact its performance: the number of sketches and the rows and columns in each sketch. These parameters are challenging to tune because the choice of one affects the others. If the number of columns is too large, it reduces the number of rows that will fit on the target, and the sketch may not be fully cleaned before rotating. Conversely, too many rows requires less columns and smaller sketches. As a sketch gets smaller, it becomes less accurate.

The objective of Conquest is to identify the packets responsible for queue build-up as accurately as possible. For comparison with the original evaluation, we quantify accuracy using the F-score[1], which depends on both precision and recall.

The original evaluation of Conquest found that it could achieve both precision

[1]Specifically, the cost is 1 minus the F-score

and recall greater than 90%, i.e., an F-score >90%. Parasol found a comparable configuration with an F-score of 92% (precision of 97% and recall of 87%). The Parasol optimizer used the Bayesian search strategy, and the configuration was found after 9 iterations.

The choice of metric used for cost affects the configuration chosen by the optimizer. F-score incorporates both precision and recall. A configuration with lower precision has more false positives, and a lower recall means more false negatives. Some applications may be more tolerant to false negatives, and others may prefer false positives. We can tailor the objective function based on an application's preference.

To minimize false positives, we can optimize for precision. This will result in a larger sketch, that keeps more accurate counts for each flow. On the other hand, we can optimize for recall to minimize false negatives. This produces a configuration with a smaller sketch, which will result in more flows being identified as significant contributors. In other words, more true positives, at the cost of more false positives as well.

Starflow. Starflow [74] is a telemetry system that partitions query processing between the data plane and software. The switch selects and groups per-packet records, which are sent to software for flow-level analytics (e.g., classifying traffic, identifying microbursts). Packet records are stored within buffers on the switch, and are evicted to software when their buffer is filled, no buffer is available, or there is a collision. There are two kinds of buffers, whose sizes must be configured at compile time: a "narrow" buffer, which tracks many small flows, and a "wide" buffer for tracking a few large flows.

The most important performance metric for Starflow is its eviction ratio: the ratio of flushed cache records to packets. A lower eviction ratio is preferable because it means that more packets are being covered by each record that the server must

process, saving both bandwidth and processing time at server.

The original, hand-optimized P4 code achieved an eviction ratio between 7.1% and 25%, depending on the size of the cache and the workload. The Parasol optimizer achieved an eviction ratio of 15%, well within the performance range of the original program. In other words, 15 out of every 100 packets are recirculated to evict a record from the cache. The best compiling configuration was found after 7 (out of 85) iterations (1.5 min) of simulated annealing. We found that both the sizes of the narrow and wide buffers impacted the eviction ratio. Our optimizer found, for our representative traffic trace, that a narrow cache smaller than 1024 slots and a wide cache smaller than 8192 slots resulted in an eviction ratio greater than 40%, with fixed wide and narrow caches, respectively.

Bibliography

[1] 0 (2023a). Intel Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`.

[2] 0 (2023b). P4 publications. `https://p4.org/publications/`.

[3] Abhashkumar, A., Lee, J., Tourrilhes, J., Banerjee, S., Wu, W., Kang, J.-M., & Akella, A. (2017). P5: Policy-driven optimization of P4 pipeline. In *ACM Symposium on SDN Research*, SOSR '17 (pp. 136–142). New York, NY, USA: Association for Computing Machinery.

[4] Alcoz, A. G., Busse-Grawitz, C., Marty, E., & Vanbever, L. (2022). Reducing p4 language's voluminosity using higher-level constructs. In *International Workshop on P4 in Europe*, EuroP4 '22 (pp. 19–25). New York, NY, USA: Association for Computing Machinery.

[5] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, T. V., Matus, F., Pan, R., Yadav, N., & Varghese, G. (2014). CONGA: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM* (pp. 503–514).

[6] Allen, J. R., Kennedy, K., Porterfield, C., & Warren, J. (1983). Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-*

SIGPLAN Symposium on Principles of Programming Languages, POPL '83 (pp. 177–189). New York, NY, USA: Association for Computing Machinery.

[7] Anderson, C. J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., & Walker, D. (2014). NetKAT: Semantic foundations for networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 113–126).: ACM.

[8] Baldi, M. (2020). Pensando Announces P4-programmable Platform and Joins P4 Community. https://opennetworking.org/news-and-events/blog/pensando-announces-p4-programmable-platform-and-joins-p4-community/.

[9] Banks, R., Jang, S., Carr, S., Sweany, P., & Kuras, D. (1999). A code generation framework for vliw architectures with partitioned register banks. *Procs. of 3rd. Int. Conf. on Massively Parallel Computing Systems*.

[10] Ben-Basat, R., Chen, X., Einziger, G., & Rottenstreich, O. (2018). Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE International Conference on Network Protocols* (pp. 313–323).

[11] Benson, T., Akella, A., & Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM Internet Measurement Conference*, IMC '10 (pp. 267–280). New York, NY, USA: Association for Computing Machinery.

[12] Birge-Lee, H., Apostolaki, M., & Rexford, J. (2022). It takes two to tango: Cooperative edge-to-edge routing. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22 (pp. 174–180). New York, NY, USA: Association for Computing Machinery.

[13] Blackfire Technology (2023). https://www.impactcybertrust.org/.

[14] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., & Walker, D. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 87–95.

[15] Bradley, A. R., Manna, Z., & Sipma, H. B. (2006). What's Decidable About Arrays? In E. A. Emerson & K. S. Namjoshi (Eds.), *Verification, Model Checking, and Abstract Interpretation* (pp. 427–442). Berlin, Heidelberg: Springer Berlin Heidelberg.

[16] Bryant (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 677–691.

[17] Butun, I., Tuncel, Y. K., & Oztoprak, K. (2021). Application layer packet processing using PISA switches. *Sensors*, 21(23), 8010.

[18] CAIDA (2019). CAIDA 2016 Chicago direction a traces. `https://www.caida.org/catalog/datasets/monitors/passive-equinix-chicago/`.

[19] Caprolu, M., Raponi, S., & Pietro, R. (2019). Fortress: An efficient and distributed firewall for stateful data plane sdn. *Security and Communication Networks*.

[20] Center for Infrastructure Assurance and Security (2023). National collegiate cyber defense competition.

[21] Chadha, R., Bowen, T., Chiang, C.-Y. J., Gottlieb, Y. M., Poylisher, A., Sapello, A., Serban, C., Sugrim, S., Walther, G., Marvel, L. M., et al. (2016). Cybervan: A cyber security virtual assured network testbed. In *MILCOM 2016-2016 IEEE Military Communications Conference* (pp. 1125–1130).: IEEE.

[22] Chen, P., Wu, Y., Yang, T., Jiang, J., & Liu, Z. (2021). Precise error estimation for sketch-based flow measurement. In *ACM SIGCOMM Internet Measurement Conference* (pp. 113–121).

[23] Chen, X., Feibish, S. L., Koral, Y., Rexford, J., Rottenstreich, O., Monetti, S. A., & Wang, T.-Y. (2019). Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19 (pp. 15–29). New York, NY, USA: Association for Computing Machinery.

[24] Chen, X., Kim, H., Aman, J. M., Chang, W., Lee, M., & Rexford, J. (2020). Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, SPIN '20 (pp. 35–41). New York, NY, USA: Association for Computing Machinery.

[25] Cooper, K. (1998). Live range splitting in a graph coloring register allocator. *International Conference on Compiler Construction*.

[26] Cormode, G. & Muthukrishnan, S. (2005). An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1), 58–75.

[27] de Moura, L. & Bjørner, N. (2008). Z3: An Efficient SMT Solver. In C. R. Ramakrishnan & J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340). Berlin, Heidelberg: Springer Berlin Heidelberg.

[28] DeLine, R. & Fahndrich, M. (1999). Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*.

[29] Fisher, J. A. (1983). Very long instruction word architectures and the eli-512. In *Proceedings of the 10th Annual International Symposium on Computer Archi-*

tecture, ISCA '83 (pp. 140–150). New York, NY, USA: Association for Computing Machinery.

[30] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., & Walker, D. (2011). Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming* (pp. 279–291).: ACM.

[31] Gao, J., Zhai, E., Liu, H. H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., & Yu, M. (2020a). Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM* (pp. 435–450).

[32] Gao, X., Kim, T., Wong, M. D., Raghunathan, D., Varma, A. K., Kannan, P. G., Sivaraman, A., Narayana, S., & Gupta, A. (2020b). Switch code generation using program synthesis. In *ACM SIGCOMM* (pp. 44–61).

[33] Gao, X., Raghunathan, D., Fang, R., Wang, T., Zhu, X., Sivaraman, A., Narayana, S., & Gupta, A. (2023). Cat: A solver-aided compiler for packet-processing pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023 (pp. 72–88). New York, NY, USA: Association for Computing Machinery.

[34] Gifford, D. K. & Lucassen, J. M. (1986). Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86 (pp. 28–38). New York, NY, USA: Association for Computing Machinery.

[35] Girard, J.-Y. (1987). Linear logic. *Theor. Comput. Sci.*, 50(1), 1–102.

[36] Hogan, M., Landau-Feibish, S., Arashloo, M. T., Rexford, J., & Walker, D. (2022). Modular switch programming under resource constraints. In *USENIX Symposium on Networked Systems Design and Implementation* (pp. 193–207). Renton, WA: USENIX Association.

[37] Hogan, M., Loehr, D., Sonchack, J., Landau-Feibish, S., Walker, D., & Rexford, J. (In Submission). Automated optimization of parameterized data-plane programs. *In Submission*.

[38] Hsu, K.-F., Beckett, R., Chen, A., Rexford, J., & Walker, D. (2020). Contra: A programmable system for performance-aware routing. In *USENIX Symposium on Networked Systems Design and Implementation* (pp. 701–721).

[39] Ibanez, S., Antichi, G., Brebner, G., & McKeown, N. (2019). Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19 (pp. 133–140). New York, NY, USA: Association for Computing Machinery.

[40] Igarashi, A. & Kobayashi, N. (2005). Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2), 264–313.

[41] Jin, X., Li, X., Zhang, H., Soule, R., Lee, J., Foster, N., Kim, C., & Stoica, I. (2017). NetCache: Balancing key-value stores with fast in-network caching. In *Symposium on Operating System Principles*.

[42] Jose, L., Yan, L., Varghese, G., & McKeown, N. (2015). Compiling packet programs to reconfigurable switches. In *USENIX Conference on Networked Systems Design and Implementation* (pp. 103–115). USA: USENIX Association.

[43] JuniperNetworks (2023). Fib compression - optimizing your routing tables.

[44] Karpilovsky, E., Caesar, M., Rexford, J., Shaikh, A., & van der Merwe, J. (2012). Practical Network-Wide Compression of IP Routing Tables. *IEEE Transactions on Network and Service Management*, 9(4), 446–458.

[45] Katta, N., Hira, M., Kim, C., Sivaraman, A., & Rexford, J. (2016). Hula: Scalable load balancing using programmable data planes. In *ACM SIGCOMM Symposium on SDN Research* (pp. 1–12).

[46] Kiselyov, Oleg (2022). How ocaml type checker works – or what polymorphism and garbage collection have in common. `http://okmij.org/ftp/ML/generalization.html`.

[47] Kuka, M., Vojanec, K., Kučera, J., & Benáček, P. (2019). Accelerated DDoS Attacks Mitigation using Programmable Data Plane. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (pp. 1–3).

[48] Lantz, B., Heller, B., & McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks* (pp. 1–6).

[49] Li, Y., Gao, J., Zhai, E., Liu, M., Liu, K., & Liu, H. H. (2022). Cetus: Releasing P4 programmers from the chore of trial and error compiling. In *USENIX Symposium on Networked Systems Design and Implementation* (pp. 371–385). Renton, WA: USENIX Association.

[50] Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., & Sekar, V. (2021). Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security Symposium*.

[51] Loehr, D. & Walker, D. (2022). Safe, modular packet pipeline programming. *Proc. ACM Program. Lang.*, 6(POPL).

[52] Mehta, V., Loehr, D., Sonchack, J., & Walker, D. (2023). Switchlog: A logic programming language for network switches. In *Proceedings of the 25th International Symposium on Practical Aspects of Declarative Languages*, PADL '23 (pp. 180–196).

[53] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 348–375.

[54] Namkung, H., Kim, D., Sekar, V., & Steenkiste, P. (2022). Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In *USENIX NSDI 2022*: USENIX.

[55] Naor, M. & Yogev, E. (2013). Sliding Bloom Filters. In L. Cai, S.-W. Cheng, & T.-W. Lam (Eds.), *Algorithms and Computation* (pp. 513–523). Berlin, Heidelberg: Springer Berlin Heidelberg.

[56] Nelson, T., Ferguson, A. D., Scheer, M. J., & Krishnamurthi, S. (2014). Tierless programming and reasoning for software-defined networks. In *USENIX Networked Systems Design and Implementation* (pp. 519–531).

[57] OCaml (2023). Data types and matching. `https://ocaml.org/docs/data-types`. Accessed: 5/24/2023.

[58] P4 Language Consortium (2018a). $P4_{14}$ language specifications.

[59] P4 Language Consortium (2018b). $P4_{16}$ language specifications.

[60] Polakow, J. & Pfenning, F. (1999). Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*.

[61] Pous, D. (2015). Symbolic algorithms for language equivalence and kleene algebra with tests. *SIGPLAN Not.*, 50(1), 357–368.

[62] Reich, J., Monsanto, C., Foster, N., Rexford, J., & Walker, D. (2013). Modular sdn programming with pyretic. *Technical Reprot of USENIX*, (pp.30).

[63] Rétvári, G., Tapolcai, J., Kőrösi, A., Majdán, A., & Heszberger, Z. (2013). Compressing ip forwarding tables: Towards entropy bounds and beyond. *SIGCOMM Comput. Commun. Rev.*, 43(4), 111–122.

[64] Rifai, M., Huin, N., Caillouet, C., Giroire, F., Pacheco, D. L., Moulierac, J., & Urvoy-Keller, G. (2014). Too Many SDN Rules? Compress Them with MINNIE. *2015 IEEE Global Communications Conference (GLOBECOM)*, (pp. 1–7).

[65] Riley, G. F. & Henderson, T. R. (2010). The ns-3 network simulator. In *Modeling and tools for network simulation* (pp. 15–34). Springer.

[66] Sánchez, J. & González, A. (2000). Instruction scheduling for clustered vliw architectures. In *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00 (pp. 41–46). USA: IEEE Computer Society.

[67] Schlesinger, C., Greenberg, M., & Walker, D. (2014). Concurrent NetCore: From policies to pipelines. In *ACM SIGPLAN International Conference on Functional programming* (pp. 11–24).: ACM.

[68] Sengupta, S., Kim, H., & Rexford, J. (2022). Continuous in-network round-trip time monitoring. In *ACM SIGCOMM*, SIGCOMM '22 (pp. 473–485). New York, NY, USA: Association for Computing Machinery.

[69] Shin, S. W., Porras, P., Yegneswara, V., Fong, M., Gu, G., & Tyson, M. (2013). Fresco: Modular composable security services for software-defined networks. In *Network & Distributed System Security Symposium*.

[70] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., & Licking, S. (2016). Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM* (pp. 15–28).

[71] Smolka, S., Eliopoulos, S., Foster, N., & Guha, A. (2015). A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015 (pp. 328–341). New York, NY, USA: Association for Computing Machinery.

[72] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems* (pp. 404–415).

[73] Sonchack, J., Loehr, D., Rexford, J., & Walker, D. (2021). Lucid: A language for control in the data plane. In *ACM SIGCOMM*, SIGCOMM '21 (pp. 731–747). New York, NY, USA: Association for Computing Machinery.

[74] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., & Smith, J. M. (2018). Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow. In *USENIX Annual Technical Conference* (pp. 823–835).

[75] Sultana, N., Sonchack, J., Giesen, H., Pedisich, I., Han, Z., Shyamkumar, N., Burad, S., DeHon, A., & Loo, B. T. (2021). Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX Symposium on Networked Systems Design and Implementation* (pp. 571–592).

[76] Tofte, M. & Birkedal, L. (1998). A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4), 724–767.

[77] Tofte, M. & Talpin, J.-P. (1997). Region-based memory management. *Inf. Comput.*, 132(2), 109–176.

[78] Vanini, E., Pan, R., Alizadeh, M., Taheri, P., & Edsall, T. (2017). Let it flow: Resilient asymmetric load balancing with flowlet switching. In *USENIX Symposium on Networked Systems Design and Implementation* (pp. 407–420). Boston, MA: USENIX Association.

[79] Vass, B., Bérczi-Kovács, E., Raiciu, C., & Rétvári, G. (2020). Compiling packet programs to reconfigurable switches: Theory and algorithms. In *P4 Workshop in Europe* (pp. 28–35).

[80] Veksler, V. D., Buchler, N., Hoffman, B. E., Cassenti, D. N., Sample, C., & Sugrim, S. (2018). Simulations in cyber-security: A review of cognitive modeling of network attackers, defenders, and users. *Frontiers in Psychology*, 9, 691.

[81] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., & Hudak, P. (2013). Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM*, volume 43 (pp. 87–98).

[82] Wette, P., Dräxler, M., Schwabe, A., Wallaschek, F., Zahraee, M. H., & Karl, H. (2014). Maxinet: Distributed emulation of software-defined networks. In *IFIP Networking Conference* (pp. 1–9).: IEEE.

[83] Wintermeyer, P., Apostolaki, M., Dietmüller, A., & Vanbever, L. (2020). P2GO: P4 profile-guided optimizations. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets '20 (pp. 146–152). New York, NY, USA: Association for Computing Machinery.

[84] Yoo, S., Chen, X., & Rexford, J. (2024). Smartcookie: Blocking large-scale syn floods with a split-proxy defense on programmable data planes. *USENIX Security*.

[85] Yu, L., Sonchack, J., & Liu, V. (2022). OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination. In *Symposium on Networked Systems Design and Implementation*.

[86] Zeno, L., Ports, D. R. K., Nelson, J., & Silberstein, M. (2020). SwiShmem: Distributed shared state abstractions for programmable switches. In *ACM SIGCOMM HotNets Workshop*.

[87] Zhang, Q., Ng, K. K., Kazer, C., Yan, S., Sedoc, J., & Liu, V. (2021). MimicNet: Fast performance estimates for data center networks with machine learning. In *ACM SIGCOMM* (pp. 287–304).

[88] Zhang, X., Wu, H., & Xue, J. (2011). An efficient heuristic for instruction scheduling on clustered vliw processors. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11 (pp. 35–44). New York, NY, USA: Association for Computing Machinery.

[89] Zheng, Y., Chen, X., Braverman, M., & Rexford, J. (2022a). Unbiased delay measurement in the data plane. In *SIAM Symposium on Algorithmic Principles of Computer Systems*.

[90] Zheng, Y., Yu, H., & Rexford, J. (2022b). Detecting TCP Packet Reordering in the Data Plane. *ArXiv*, abs/2301.00058.

[91] Zhou, Z., Lv, J., Cheng, L., Chen, X., Zhang, T., Huang, Q., Luo, J., Zhu, L., Zhang, D., & Wu, C. (2022). SketchGuide: Reconfiguring sketch-based measurement on programmable switches. In *IEEE International Conference on Network Protocols (ICNP)* (pp. 1–11).