Abstractions for Software-defined Networks

Cole Nathan Schlesinger

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

June 2015

Abstract

In a Software-Defined Network (SDN), a central, computationally powerful *controller* manages a set of distributed, computationally simple *switches*. The controller computes a policy describing how each switch should route packets and *populates* packet-processing tables on each switch with rules to enact the routing policy. As network conditions change, the controller continues to add and remove rules from switches to adjust the policy as needed.

Recently, the SDN landscape has begun to change as several proposals for new, reconfigurable switching architectures, such as RMT [6] and FlexPipe [42], have emerged. These platforms provide switch programmers with many flexible tables for storing packet-processing rules, and they offer programmers control over the packet fields that each table can analyze and act on. These reconfigurable switch architectures support a richer SDN model in which a switch *configuration* phase precedes the rule population phase [5]. In the configuration phase, the controller sends the switch a graph describing the layout and capabilities of the packet processing tables it will require during the population phase. Armed with this foreknowledge, the switch can allocate its hardware (or software) resources more efficiently.

This dissertation presents a new, typed language, called Concurrent NetCore, for specifying routing policies *and* graphs of packet-processing tables. Concurrent NetCore includes features for specifying sequential, conditional, and concurrent control-flow between packet-processing tables. We develop a fine-grained operational model for the language and prove this model coincides with a higher-level denotational model when programs are well-typed. We also prove several additional properties of well-typed programs, including strong normalization and determinism. To illustrate the utility of the language, we develop linguistic models of both the RMT and FlexPipe architectures; give a multi-pass compilation algorithm that translates graphs and

routing policies to the RMT model; and evaluate a prototype of the language and compiler on two benchmark applications, a learning switch and a stateful firewall.

Acknowledgements

Many people have helped make this dissertation possible, but none more than my adviser. Dave, thank you for your guidance and encouragement throughout the many years of my studies.

I thank my readers, Arjun Guha and Aarti Gupta, for their heroic efforts in reviewing this dissertation as well as their insightful comments, and my examiners, Nick Feamster and Jen Rexford, for their feedback on the journey from research to writing. The result is all the stronger for it.

I have worked with wonderful collaborators at Princeton and elsewhere, including: Michael Greenberg and Jen Rexford, both at Princeton, Nate Foster, Arjun Guha, Dexter Kozen, Jean-Baptiste Jeannin, Carolyn Jane Anderson, Mark Reitblatt, and Alec Story, all at Cornell, Marco Canini at UC Louvain, Nikhil Swamy, Ben Zorn, Juan Chen, Ben Livshits, and Joel Weinberger at MSR Redmond, Hitesh Ballani, Thomas Karagiannis, and Dimitrios Vytiniotis at MSR Cambridge, and Soudeh Ghorbani and Matt Caesar at UIUC. Thank you for your insight, guidance, and energy. I am especially grateful to Nate, for his indispensable advice throughout my studies, and to Michael, for his technical insight into our work and his encouragement to set it aside in favor of climbing.

Tim Teitelbaum and the folks at GrammaTech helped set me on the road to graduate school, for which I am most appreciative. And the PL, Systems, and Security groups at Princeton have had an indelible impact on my growth as a researcher. Thank you for the many interesting and thought-provoking discussions. To Gordon Stewart and Josh Kroll, who began and will soon end this journey with me: Best of luck as we continue on.

There are many pitfalls that make life as a graduate student more challenging; thankfully, I avoided most of them with help from Michele Brown, Pamela DelOrefice, Nicki Gotsis, Mitra Kelly, Melissa Lawson, Barbara Varga, and Nicole Wagenblast.

Thank you. And I will always be grateful to Laura Cummings-Abdo for feeding my coffee addiction, and to the fine folks at España for fueling the production of my dissertation with delicious sandwiches and tapas.

And, as a final acknowledgement, Dave's grant has my undying gratitude for the heavy load it was made to bear.

To my parents and sisters.

Contents

List of Figures

Chapter 1

Introduction

Traditional networks are complex constructs built from special-purpose devices like switches, routers, firewalls, load balancers, and middle-boxes. Such devices often support standard protocols for transporting data but come with proprietary interfaces for configuration and control. Indeed, network administrators have been called "masters of complexity" for their ability to reconcile the intricate configuration details of different devices with different interfaces from different vendors in order to bring forth correct system-wide behavior [46]. Even still, configuration errors are responsible for high profile outages of services like Amazon EC2 and Microsoft Azure [49, 39]. This ad-hoc design makes it difficult to safely extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, in recent years, a revolution has taken place with the rise of *software-defined networking* (SDN). In SDN, a general-purpose *controller machine* manages a collection of *programmable switches*. The controller responds to network events such as newly connected hosts, topology changes, and shifts in traffic load by reprogramming the switches accordingly. This *logically centralized*, global view of the network makes it possible to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophis-

ticated applications such as load balancing, intrusion detection, and fault-tolerance using commodity hardware.

SDN has had a tremendous impact in the networking community, both for industry and academia. Google has adopted SDN to manage its internal backbone, which transmits all its intra-datacenter traffic—making it one of the largest networks in the world [17], and many other major companies are following Google's lead. Indeed, the board of the Open Networking Foundation (ONF)—the main body responsible for defining SDN standards, such as OpenFlow [34]—includes the owners of most of the largest networks in the world (Google, Facebook, Microsoft, *etc.*) and its corporate membership numbers over a hundred. On the academic side, hundreds of participants have attended the newly-formed HotSDN workshop, and several tracks of top networking conferences, such as NSDI and SIGCOMM, are dedicated to research in SDN. But at its heart, management of SDNs is an important new programming problem that calls for a variety of new, high-level, declarative, domain-specific programming languages, as well as innovation in compiler design and implementation.

Software-defined networking represents open, programmatic control of networking devices, and recent proposals from both academia and industry have called for moving beyond the fixed-protocol processing embodied in the initial OpenFlow standards toward *protocol-independent* packet processing [5, 48], wherein reconfigurable switching hardware can be configured to extract and operate on arbitrary packet header fields rather than fixed protocols. Indeed, hardware platforms are emerging to provide such flexibility while balancing latency, cost, and power consumption [6, 42].

This dissertation presents a high-level language for specifying switch pipelines and packet forwarding policies, along with linguistic models of concrete switch architectures, and compilation algorithms for compiling from one to the other. The remainder of this chapter introduces background material on software-defined networks, network policy languages, and reconfigurable switching architectures.

1.1 The Architecture of an SDN

Figure 1.1 illustrates the basic architecture of a software-defined network, wherein a logically centralized *controller* machine (or cluster of machines) manages a distributed collection of *switches*. The controller is a general-purpose server whose primary job is to decide how to route packets through the network while avoiding congestion, managing security, handling failures, monitoring load, and informing network operators of problems. The switches, on the other hand, are specialized hardware devices with limited computational facilities. In general, a switch implements a collection of simple rules that match bit patterns in the incoming packets, and based on those bit patterns, drop packets, modify their fields, forward the packets on to other switches, or send the packet to the controller for additional, more general analysis and processing. The switch itself does not decide what rules to implement—that job lies with the controller, which sends messages to the switches to install and uninstall the packet-forwarding rules needed to achieve its higher-level, network-wide objectives. SDN is distinguished from traditional networks by its centralized, programmatic control. In contrast, traditional networks rely on distributed algorithms implemented by the switches, and network administrators manually configure a combination of distributed protocols and local control logic on each switch in the hope of inducing behavior that conforms to a global (and often poorly specified) network policy.

As with any software system, the capabilities of the hardware are only available to the controller insofar as the interface (or protocol) makes them available. At the same time, a software stack, coupled with a run-time system, can build stronger abstractions atop lower-level interfaces. The capabilities of both the hardware and software abstractions have evolved inter-dependently in the SDN community in recent years, a trend especially visible in the evolution of the OpenFlow protocol.

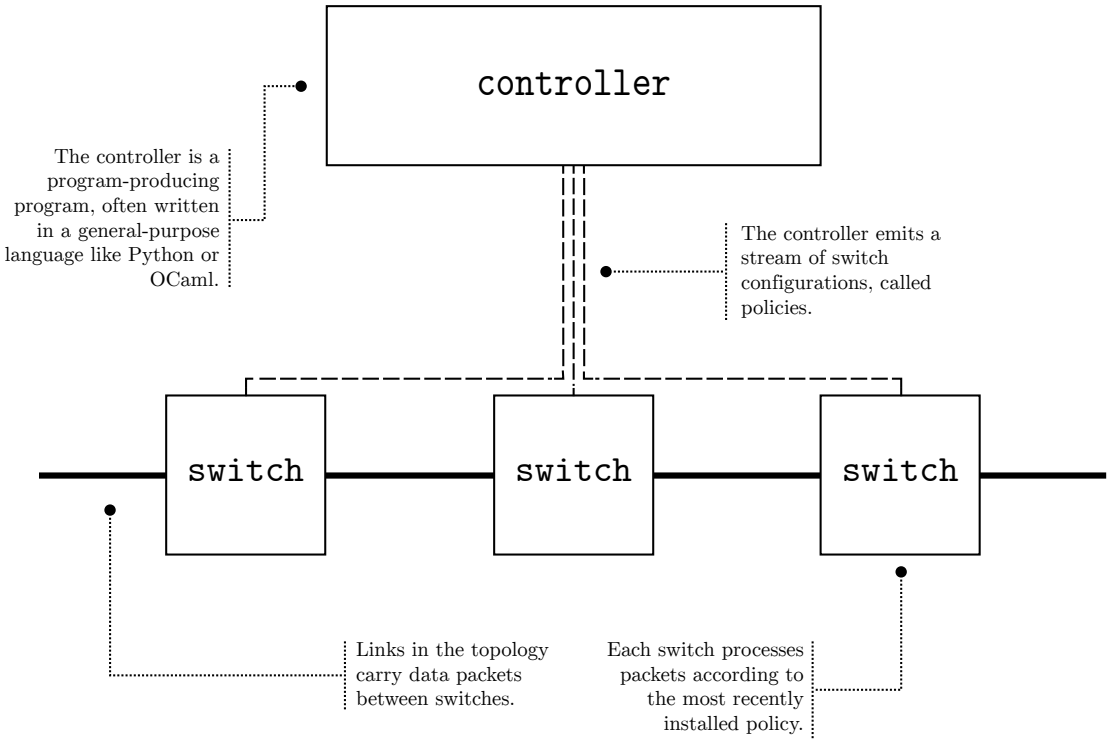

Figure 1.1: The architecture of a software-defined network.

OpenFlow 1.0: successes and failures. In OpenFlow 1.0, each switch is a *single* table of packet-forwarding rules. Each such rule can match on one or more of twelve standard packet fields (source MAC, destination MAC, source IP, destination IP, VLAN, *etc.*) and then execute a series of actions, such as dropping the packet, modifying a field, or forwarding it out a port. A controller can issue commands to install and uninstall rules in the table and to query statistics associated with each rule (*e.g.,* the number of packets or bytes processed).

The single table abstraction was chosen for the first version of OpenFlow because it was a "least common denominator" interface that many existing switches could support with little change. It worked, and OpenFlow switches from several hardware vendors, including Broadcom and Intel, hit the market quickly. The simplicity of the OpenFlow 1.0 interface also made it a relatively easy compilation target for a wave

of newly-designed, high-level SDN programming languages, such as Frenetic [11], Procera [53], Maple [54], FlowLog [40] and others.

Unfortunately, while the simplicity of the OpenFlow 1.0 interface is extremely appealing, hardware vendors have been unable to devise implementations that make efficient use of switch resources. Packet processing hardware in most modern ASICs is not, in fact, implemented as a single match-action table, but rather as a collection of tables. These tables are often aligned in sequence, so the effects of packet processing by one table can be observed by later tables, or in parallel, so non-conflicting actions may be executed concurrently to reduce packet-processing latency.

Each table within a switch will typically match on a fixed subset of a packet's fields and will be responsible for implementing some subset of the chip's overall packet-forwarding functionality. Moreover, different tables may be implemented using different kinds of memory with different properties. For example, some tables might be built with SRAM and only capable of *exact matches* on certain fields—that is, comparing fields against a single, concrete bit sequence (*e.g.* 1010001010). Other tables may use TCAM and be capable of *ternary wildcard matches*, where packets are compared to a string containing concrete bits and wildcards (*e.g.* 10?1??1001?) and the wildcards match either 0 or 1. TCAM is substantially more expensive and power-hungry than SRAM. Hence, TCAM tables tend to be smaller than SRAM. For instance, the Broadcom Trident has an L2 table with SRAM capable of holding ∼100K entries and a forwarding table with TCAM capable of holding ∼4K entries [7].

In addition to building fixed-pipeline ASICs, switch hardware vendors are also developing more programmable hardware pipelines. For example, the RMT design [6] offers a programmable parser to extract data from packets in arbitrary application-driven ways, and a pipeline of 32 physical match-action tables. Each physical table in this pipeline may be configured for use in different ways: (1) As a wide table, matching many bits at a time, but containing fewer rows, (2) as a narrower table, matching fewer

bits in each packet but containing more rows, (3) as multiple parallel tables acting concurrently on a packet, or (4) combined with other physical tables in sequence to form a single, multi-step logical table. Intel's FlexPipe architecture [42] also contains a programmable front end, but rather than organizing tables in a sequential pipeline, FlexPipe contains a collection of parallel tables to allow concurrent packet processing, a shorter pipeline and reduced packet-processing latency.

In theory, these multi-table hardware platforms could be programmed through the single-table OpenFlow 1.0 interface. However, doing so has several disadvantages:

- The single OpenFlow 1.0 interface serves as a bottleneck in the compilation process: Merging rules from separate tables into a single table can lead to an explosion in the number of rules required to represent the same function as one might represent via a set of tables.

- Once squeezed into a single table, the structure of the rule set is lost. Recovering that structure and determining how to split rules across tables is a non-trivial task, especially when the rules appear dynamically (without advance notice concerning their possible structure) at the switch.

- Newer, more flexible chips such as RMT, FlexPipe or NetFPGAs have a configuration stage, wherein one plans the configuration of tables and how to allocate different kinds of memory. The current OpenFlow protocol does not support configuration-time planning.

Towards OpenFlow 2.0. As a result of the deficiencies of the first generation of OpenFlow protocols, a group of researchers have begun to define an architecture for the next generation of OpenFlow protocols [5] (See Figure 1.2). In this proposal, switch configuration is divided into two phases: table configuration and table population.

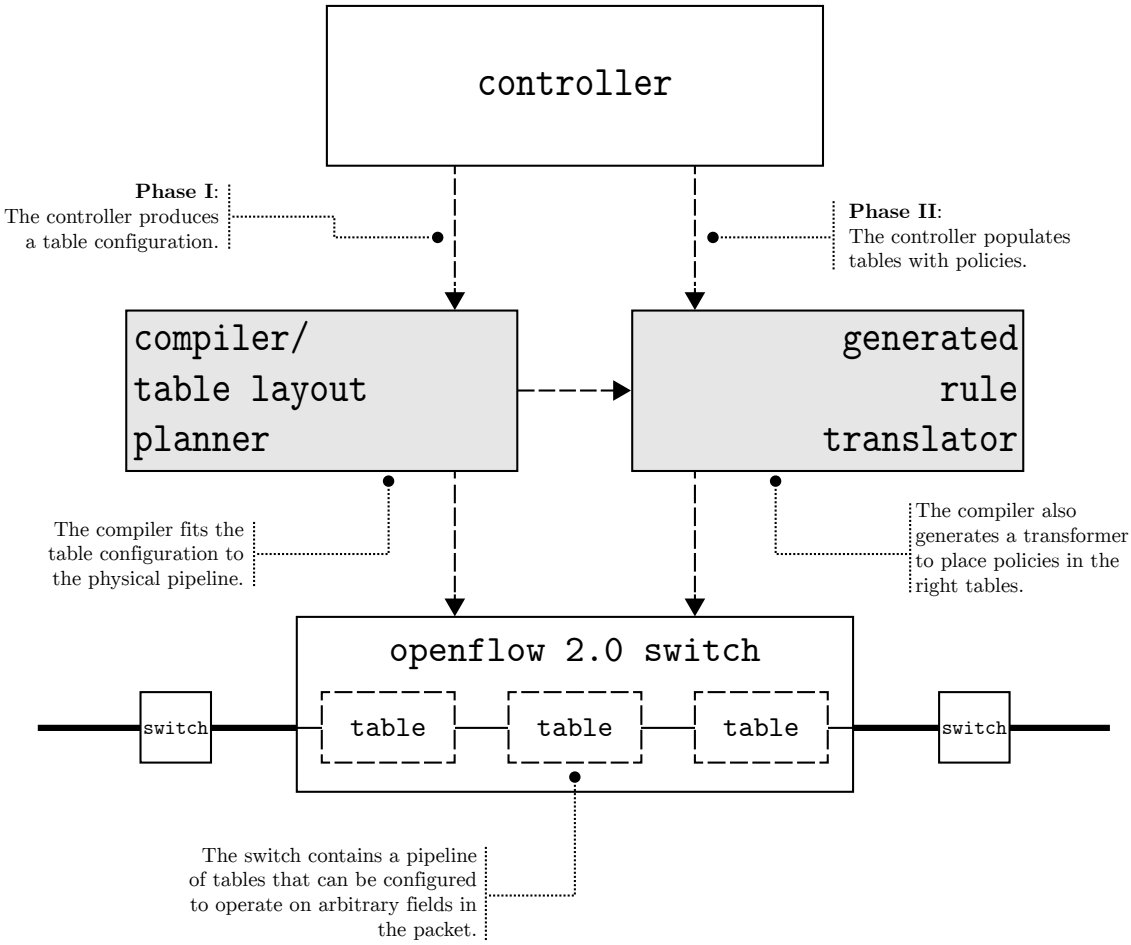

Figure 1.2: The architecture of an "OpenFlow 2.0" software-defined network.

During the table configuration phase, the SDN controller describes the *abstract* set of tables it requires for its high-level routing policy. When describing these tables, it specifies the packet fields read and written by each table, and the sorts of patterns (either exact match or prefix match) that will be used. In addition, the table configuration describes the topology of the abstract tables—the order they appear in sequence (or in parallel) and the conditions, if any, on branches in the pipeline that dictate when the rules within a table will be applied to a given packet.

We call the tables communicated from controller to switch *abstract*, because they do not necessarily correspond directly to the *concrete* physical tables implemented by the switch hardware. In order to bridge the gap between abstract and concrete tables,

a compiler will attempt to find a mapping between what is requested by the controller and what is present in hardware. Figure 1.2 shows the compiler/table layout planner as residing between the controller and the switches, because the controller logic is defined in terms of abstract tables. But we expect that, in practice, the compiler will map abstract to concrete tables as part of the controller infrastructure.

In the process of determining this mapping, the compiler will generate a function capable of translating sets of *abstract rules* (also called an *abstract policy*) supplied by the controller, and targeted at the abstract tables, into *concrete rules/policy* implementable directly on the concrete tables available in hardware. After the table configuration phase, and during the table population phase, the rule translator is used to transform abstract rules into concrete ones.

The configuration phase happens on a human time scale: a network administrator writes a policy and a controller program and runs the compiler to configure the switches and SDN controllers on her network appropriately. Rule population, on the other hand, happens on the time scale of network activity: a controller's algorithm may install, e.g., new firewall or NAT rules after observing a single packet—concrete examples of these and other rule installations can be found in Chapter 2.

1.2 Contributions of this Dissertation

The central contribution of this dissertation is the design of a new language for programming "OpenFlow 2.0"-style switches. This compile intermediate language is capable of specifying high-level switch policies as well as concrete, low-level switch architectures. We call the language *Concurrent NetCore* (or CNC, for short), as it is inspired by past work on NetCore [11, 37] and NetKAT [2].[1] Like NetCore and NetKAT, Concurrent NetCore consists of a small number of primitive operations for

[1]Readers familiar with NetKAT and NetCore will note that our language does not contain Kleene star, which is more useful for specifying paths across a network than policies on a single switch. Hence, our language is a Net*Core* as opposed to a Net*KAT*.

specifying packet processing, plus combinators for constructing more complex packet processors from simpler ones. Concurrent NetCore introduces the following features.

- *Table specifications*: Table specifications act as "holes" in an otherwise fully-formed switch policy. These tables can be filled in (*i.e.,* populated) later. Policies with tables serve as the phase-1 configurations in the OpenFlow 2.0 architecture. Ordinary, hole-free policies populate those holes later in the switch-configuration process.

- *Concurrent composition*: Whereas NetCore and NetKAT have a form of "parallel composition," which copies a packet and performs different actions on different copies, CNC also provides a new *concurrent* composition operator that allows two policies to act simultaneously on the same packet. We use concurrent composition along with other features of CNC to model the RMT and Intel FlexPipe packet-processing pipelines.

- *Type System*: Unlike past network programming languages, CNC is equipped with a simple domain-specific type system. These types perform two functions: (1) they determine the kinds of policies that may populate a table (which fields may be read or written, for instance), and thereby guarantee that well-typed policies can be compiled to the targeted table, and (2) they prevent interference between concurrently executing policies, thereby ensuring that the overall semantics of a CNC program is deterministic.

The key technical results include the following:

- *Semantics for Concurrent NetCore*: We define a small-step operational semantics for CNC that captures the intricate interactions between (nested) concurrent and parallel policies. In order to properly describe interacting concurrent actions, this semantics is structured entirely differently from the denotational models previously defined for related languages.

- *Metatheory of Concurrent NetCore*: The metatheory includes a type system and its proof of soundness, as well as several auxiliary properties of the system, such as confluence and normalization of all well-typed policies. We derive reasoning principles relating the small-step CNC semantics to a NetKAT-like denotational model.

- *Multipass compilation algorithm*: We show how to compile high-level *abstract* configurations into the constrained lower-level *concrete* configuration of the RMT pipeline [6]. In doing so, we show how to produce *policy transformation functions* that will map abstract policy updates into concrete policy updates. We have proven many of our compilation passes correct using reasoning principles derived from our semantics. We offer this compilation as a proof of concept of "transformations within CNC" as a compilation strategy; we believe that many of our algorithms and transformations will be reusable when targeting other platforms.

The remainder of this dissertation is structured as follows. Chapter 2 presents background material on the syntax and semantics of NetKAT, as well as an application of the equational theory in the form of a slice compilation algorithm for inducing network isolation. Chapter 3 extends the switch-local fragment of NetKAT for reasoning about switch pipelines, with Section 3.2 presenting linguistic models of three pipeline architectures. Chapter 4 presents a semantics-preserving single-table compilation algorithm (an extension of the algorithm in [2]) and goes on to describe compilation algorithms for a full pipeline of tables. Chapter 5 implements and evaluates the algorithms found in Chapter 4; Chapter 6 explores related work in greater depth; and Chapter 7 concludes with a discussion of open questions and avenues for future work.

1.3 Relation to Previous Work by the Author and Co-Authors

The technical contributions of this dissertation have grown out of joint work with wonderful collaborators. The NetKAT language and equational theory (Chapter 2) arose from a collaboration with Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, and David Walker, and first appeared in POPL'14 [2]. A pipeline model and configuration language, called P4, was first presented informally in [5], which I was nominally involved with as a co-author. Concurrent NetCore, which extends this language with concurrency and union operators, an operational semantics, and proved correct compilation algorithms (Chapters 3 and 4), first appeared in ICFP'14 [45] as joint work with Michael Greenberg and David Walker.

Chapter 2

A Network Policy Language

In this chapter, we explore a foundational model for network programming that (1) identifies essential constructs for programming networks, (2) provides guidelines for incorporating new features, and (3) unifies reasoning about switches, topology, and end-to-end behavior. This chapter is primarily drawn from [2].

Semantic foundations. We begin our development by focusing on the global behavior of the whole network. This is in contrast to previous languages, which have focused on the local behavior of the individual switches. Abstractly, a network can be seen as an automaton that shuttles packets from node to node along the links in its topology. Hence, from a linguistic perspective, it is natural to begin with regular expressions, the language of finite automata. Regular expressions are a natural way to specify the packet-processing behavior of a network: a path is represented as a concatenation of processing steps $(p; q; \cdots)$, a set of paths is represented using union $(p + q + \cdots)$, and iterated processing is represented using Kleene star. Moreover, by modeling networks in this way, we get a ready-made theory for reasoning about formal properties: *Kleene algebra* (KA), a decades-old sound and complete equational theory of regular expressions.

With Kleene algebra as the choice for representing global network structure, we can turn our attention to specifying local switch-processing functionality. Fundamentally, a switch implements *predicates* to match packets and *actions* that transform and forward matching packets. Existing languages build various abstractions atop the predicates and actions supplied by the hardware, but predicates and actions are essential. As a consequence, a foundational model for SDN must incorporate both *Kleene algebra* for reasoning about network structure and *Boolean algebra* for reasoning about the predicates that define switch behavior. Fortunately, these classic mathematical structures have already been unified in previous work on *Kleene algebra with tests* (KAT) [26].

By now KAT has a well-developed metatheory, including an extensive model theory and results on expressiveness, deductive completeness, and complexity. The axioms of KAT are sound and complete over a variety of popular semantic models, including language, relational, and trace models, and KAT has been applied successfully in a number of application areas, including the verification of compiler optimizations, device drivers and communication protocols. Moreover, equivalence in KAT has a PSPACE decision procedure. This paper applies this theory to a new domain: networks.

NetKAT. NetKAT is a new framework based on Kleene algebra with tests for specifying, programming, and reasoning about networks [2]. As a programming language, NetKAT has a simple denotational semantics inspired by that of NetCore [37], but modified and extended in key ways to make it sound for KAT (which NetCore is not). In this respect, the semantic foundation provided by KAT has delivered true guidance: the axioms of KAT dictate the interactions between primitive program actions, predicates, and other operators. NetKAT thus provides a foundational structure and consistent reasoning principles that other network programming languages lack.

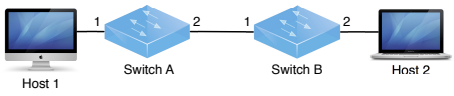

Figure 2.1: A simple example network.

For specification and reasoning, NetKAT also has an axiomatic semantics, characterized by a finite set of equations that capture equivalences between NetKAT programs. The equational theory includes the axioms of KAT, as well as domain-specific axioms that capture manipulations of packets. These axioms enable reasoning about local switch processing functionality (needed in compilation and optimization) as well as global network behavior (needed to check reachability and traffic isolation properties). The equational theory is sound and complete with respect to the denotational semantics.

NetKAT by example. This section begins by introducing the syntax and informal semantics of NetKAT with a simple example. Consider the network shown in Figure 2.1. It consists of two switches, A and B, each with two ports labeled 1 and 2, and two hosts. The switches and hosts are connected together in series. Suppose we want to configure the network to provide the following services:

- *Forwarding:* transfer packets between the hosts, but
- *Access control:* block SSH packets.

The forwarding component is straightforward—program each switch to forward packets destined for host 1 out port 1, and similarly for host 2—but there are many ways to implement the access control component. We will describe several implementations and show that they are equivalent using NetKAT's equational theory.

Forwarding. To warm up, let us define a simple NetKAT policy that implements forwarding. To a first approximation, a NetKAT policy can be thought of as a function from packets to sets of packets. (In the next section we will generalize this

type to functions from lists of packets to sets of lists of packets, where the lists encode packet-processing histories, to support reasoning about network-wide properties.) We represent packets as records with fields for standard headers such as source address (src), destination address (dst), and protocol type (typ), as well as two fields, switch (sw) and port (pt), that identify the location of the packet in the network.

The most basic NetKAT policies are filters and modification. A filter $(f = n)$ takes an input packet pk and yields the singleton set $\{pk\}$ if field f of pk equals n, and $\emptyset$ otherwise. A modification $(f \leftarrow n)$ takes an input packet pk and yields the singleton set $\{pk'\}$, where pk' is the packet obtained from pk by setting f to n.

To help programmers build richer policies, NetKAT also has policy combinators that build bigger policies out of smaller ones. Parallel composition $(p + q)$ produces the union of the sets produced by applying each of p and q to the input packet, while sequential composition $(p; q)$ first applies p to the input packet, then applies q to each packet in the resulting set, and finally takes the union of the resulting sets. Using these operators, we can implement the forwarding policy for the switches:

$$p \triangleq (\mathsf{dst} = H_1; \mathsf{pt} \leftarrow 1) + (\mathsf{dst} = H_2; \mathsf{pt} \leftarrow 2)$$

At the top level, this policy is the union of two sub-policies. The first updates the pt field of all packets destined for H_1 to 1 and drops all other packets, while the second updates the pt field of all packets destined for H_2 to 2. The union of the two generates the combined results of their behaviors—in other words, the policy forwards packets across the switches in both directions.

Access control. Next, we extend the policy with access control. The simplest way to do this is to compose a filter that blocks SSH traffic with the forwarding policy:

$$p_{ac} \triangleq \neg(\mathsf{typ} = \text{SSH}); p$$

This policy drops the input packet if its typ field is SSH and otherwise forwards it using p. Of course, a quick inspection of the network topology shows that it is unnecessary to test *all* packets at *all* places in the network in order to block SSH traffic: packets travelling between host 1 and host 2 must traverse both switches, so it is sufficient to filter only at switch A:

$$p_A \triangleq (\mathsf{sw} = A; \neg(\mathsf{typ} = \text{SSH}); p) + (\mathsf{sw} = B; p)$$

or only at switch B:

$$p_B \triangleq (\mathsf{sw} = A; p) + (\mathsf{sw} = B; \neg(\mathsf{typ} = \text{SSH}); p)$$

These policies are slightly more complicated than the original policy, but are more efficient because they avoid having to store and enforce the access control policy at both switches.

Naturally, one would prefer one of the two optimized policies. Still, given these programs, there are several questions we would like to be able to answer:

- *"Are non-*SSH *packets forwarded?"*
- *"Are* SSH *packets dropped?"*
- *"Are p_{ac}, p_A, and p_B equivalent?"*

Network administrators ask these sorts of questions whenever they modify the policy or the network itself. Note that we cannot answer them just by looking at the policies—the answers depend on the network topology. We will see how to incorporate the topology as a part of the NetKAT program next.

Topology. Given a program that describes the behavior of the switches, it is not hard to define a semantics that accounts for the topology. For example, Monsanto *et*

al. [37] define a network-wide operational relation that maps sets of packets to sets of packets by modeling the topology as a partial function from locations to locations. At each step, the semantics extracts a packet from the set of in-flight packets, applies the policy and the topology in sequence, and adds the results back in to the set of in-flight packets. However, to reason syntactically about network-wide behavior we need a *uniform representation* of policy and topology, not a separate auxiliary relation.

A network topology is a directed graph with switches as nodes and links as edges. We can model the behavior of the topology as the union of policies, one for each link in the network. Each link policy is the sequential composition of a filter that retains packets located at one end of the link and a modification that updates the sw and pt fields to the location at the other end of the link, thereby capturing the effect of sending a packet across the link. We assume that links are uni-directional, and encode bi-directional links as pairs of uni-directional links. For example, we can model the links between switches A and B with the following policy:

$$t = (\mathsf{sw} = A; \mathsf{pt} = 2; \mathsf{sw} \leftarrow B; \mathsf{pt} \leftarrow 1) + (\mathsf{sw} = B; \mathsf{pt} = 1; \mathsf{sw} \leftarrow A; \mathsf{pt} \leftarrow 2)$$

As of yet, most networks topologies cannot be automatically programmed, instead requiring a certain amount of manual intervention. Hence, policies that describe the topology can be thought of as akin to assumptions, describing how the topology *should* be configured.

Switches meet topology. A packet traverses the network in interleaved steps, processed first by a switch, then sent along the topology, and so on. In our example, if host 1 sends a non-SSH packet to host 2, it is first processed by switch A, then the link between A and B, and finally by switch B. This can be encoded by the NetKAT term $p_{ac}; t; p_{ac}$. More generally, a packet may require an arbitrary number of steps—in particular, if the network topology has a cycle. Using the Kleene star

operator, which iterates a policy zero or more times, we can encode the end-to-end behavior of the network:

$$(p_{ac}; t)^*; p_{ac}$$

Note however that this policy processes packets that enter and exit the network at arbitrary locations, including internal locations such as on the link between switches A and B. When reasoning about the network, it is often useful to restrict attention to packets that enter and exit the network at specified external locations e, rather than magically appearing on internal links:

$$e \triangleq (\mathsf{sw} = A; \mathsf{pt} = 1) + (\mathsf{sw} = B; \mathsf{pt} = 2)$$

Using this predicate, we can restrict the policy to packets sent or received by one of the hosts:

$$p_{net} \triangleq e; (p_{ac}; t)^*; p_{ac}; e$$

More generally, the input and output predicates may be distinct:

$$in; (p; t)^*; p; out$$

We call a network modeled in this way a *logical crossbar* [33], since it encodes the end-to-end processing behavior of the network (and elides processing steps on internal hops). This encoding is inspired by the model used in Header Space Analysis [23]. Section 2.1 discusses a more refined encoding that models hop-by-hop processing.

Formal reasoning. We now turn to formal reasoning problems and investigate whether the logical crossbar p_{net} correctly implements the specified forwarding and access control policies. It turns out that these questions, and many others, can be reduced to policy equivalence. We write $p \equiv q$ when p and q return the same set of packets on all inputs, and $p \leq q$ when p returns a subset of the packets returned by

q on all inputs. Note that $p \leq q$ can be treated as an abbreviation for $p + q \equiv q$. To establish that p_{net} correctly filters all SSH packets, we check the following equivalence:

$$(\mathsf{typ} = \text{SSH}; p_{net}) \equiv \mathsf{drop}$$

To establish that p_{net} correctly forwards non-SSH packets from H_1 to H_2, we check the following inclusion:

$$\begin{aligned}&(\neg(\mathsf{typ} = \text{SSH}); \mathsf{sw} = A; \mathsf{pt} = 1; \mathsf{sw} \leftarrow B; \mathsf{pt} \leftarrow 2)\\ \leq\ &(\neg(\mathsf{typ} = \text{SSH}); \mathsf{sw} = A; \mathsf{pt} = 1; p_{net}; \mathsf{sw} = B; \mathsf{pt} = 2)\end{aligned}$$

and similarly for non-SSH packets H_2 to H_1. Lastly, to establish that *some* traffic can get from port 1 on switch A to port 2 on switch B, we check the following:

$$(\mathsf{sw} = A; \mathsf{pt} = 1; p_{net}; \mathsf{sw} = B; \mathsf{pt} = 2) \not\equiv \mathsf{drop}$$

Of course, to actually check these equivalences, we need a proof system. NetKAT is designed to not only be an expressive programming language, but also one that satisfies the axioms of a Kleene algebra with tests (KAT). Moreover, by extending KAT with additional axioms that capture the domain-specific features of networks, the equational theory is complete—*i.e.*, it can answer all the questions posed in this section, and many more. The following sections present the syntax, semantics, and equational theory of NetKAT formally (Sections 2.1 and 2.2), and illustrate its effectiveness through an extended example investigating program isolation (Section 2.3). Later chapters will show how to extend NetKAT and its equational theory to reason about a more sophisticated programming model that accounts for reconfigurable packet-processing hardware.

2.1 Syntax and Semantics

This section defines the NetKAT syntax and semantics formally.

Preliminaries. A packet pk is a record with fields $f_1, \ldots, f_k$ mapping to fixed-width integers n. We assume a finite set of *packet headers*, including Ethernet and IP source and destination addresses, VLAN tag, TCP and UDP ports, along with special fields for the switch (sw), port (pt), and payload. For simplicity, we assume that every packet contains the same fields. We write $pk.f$ for the value in field f of pk, and $pk[f := n]$ for the packet obtained by updating field f of pk with n.

To facilitate reasoning about the paths a packet takes through the network, we maintain a *packet history* that records the state of each packet as it travels from switch to switch. Formally, a packet history h is a non-empty sequence of packets. We write $pk{::}\langle\rangle$ to denote a history with one element, $pk{::}h$ to denote the history constructed by prepending pk on to h, and $\langle pk_1, \ldots, pk_n\rangle$ for the history with elements pk_1 to pk_n. By convention, the first element of a history is the current packet; other elements represent older packets. We write H for the set of all histories, and $\mathcal{P}(\mathsf{H})$ for the powerset of H.

Syntax. Syntactically, NetKAT expressions are divided into two categories: predicates (a, b, c) and policies (p, q, r). Predicates include constants true (id) and false (drop), matches $(f = n)$, and negation $(\neg a)$, disjunction $(a + b)$, and conjunction $(a; b)$ operators. Policies include predicates, modifications $(f \leftarrow n)$, union $(p + q)$ and sequential $(p; q)$ composition, iteration (p^*), and a special policy that records the current packet in the history (dup). The complete syntax of NetKAT is given in Figure 2.2. By convention, $(^*)$ binds tighter than $(;)$, which binds tighter than $(+)$. Hence, $a; b + c; d^*$ is the same as $(a; b) + (c; (d^*))$.

Semantics. Semantically, every NetKAT predicate and policy denotes a function that takes a history h and produces a (possibly empty) set of histories $\{h_1, \ldots, h_n\}$.

Syntax

$$
\begin{array}{rrll}
\text{Fields} & f & ::= & f_1 \mid \cdots \mid f_k \\
\text{Packets} & pk & ::= & \{f_1 = v_1, \cdots, f_k = v_k\} \\
\text{Histories} & h & ::= & pk{::}\langle\rangle \mid pk{::}h \\
\text{Predicates} & a, b, c & ::= & \mathsf{id} \quad \textit{Identity} \\
 & & \mid & \mathsf{drop} \quad \textit{Drop} \\
 & & \mid & f = n \quad \textit{Match} \\
 & & \mid & a + b \quad \textit{Disjunction} \\
 & & \mid & a; b \quad \textit{Conjunction} \\
 & & \mid & \neg a \quad \textit{Negation} \\
\text{Policies} & p, q, r & ::= & a \quad \textit{Filter} \\
 & & \mid & f \leftarrow n \quad \textit{Modification} \\
 & & \mid & p + q \quad \textit{Parallel composition} \\
 & & \mid & p; q \quad \textit{Sequential composition} \\
 & & \mid & p^* \quad \textit{Kleene star} \\
 & & \mid & \mathsf{dup} \quad \textit{Duplication}
\end{array}
$$

Figure 2.2: Syntax of NetKAT.

Semantics

$$
\begin{aligned}
[\![p]\!] &\in \mathsf{H} \to \mathcal{P}(\mathsf{H}) \\
[\![\mathsf{id}]\!]\ h &\triangleq \{h\} \\
[\![\mathsf{drop}]\!]\ h &\triangleq \emptyset \\
[\![f = n]\!]\ (pk{::}h) &\triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h &\triangleq \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ (pk{::}h) &\triangleq \{pk[f := n]{::}h\} \\
[\![p + q]\!]\ h &\triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p; q]\!]\ h &\triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p^*]\!]\ h &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ h \\
\text{where } F^0\ h &\triangleq \{h\} \text{ and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\mathsf{dup}]\!]\ (pk{::}h) &\triangleq \{pk{::}(pk{::}h)\}
\end{aligned}
$$

Figure 2.3: Semantics of NetKAT.

Kleene Algebra Axioms

$p + (q + r) \equiv (p + q) + r$	KA-Plus-Assoc
$p + q \equiv q + p$	KA-Plus-Comm
$p + \mathsf{drop} \equiv p$	KA-Plus-Zero
$p + p \equiv p$	KA-Plus-Idem
$p; (q; r) \equiv (p; q); r$	KA-Seq-Assoc
$\mathsf{id}; p \equiv p$	KA-One-Seq
$p; \mathsf{id} \equiv p$	KA-Seq-One
$p; (q + r) \equiv p; q + p; r$	KA-Seq-Dist-L
$(p + q); r \equiv p; r + q; r$	KA-Seq-Dist-R
$\mathsf{drop}; p \equiv \mathsf{drop}$	KA-Zero-Seq
$p; \mathsf{drop} \equiv \mathsf{drop}$	KA-Seq-Zero
$\mathsf{id} + p; p^* \equiv p^*$	KA-Unroll-L
$q + p; r \leq r \Rightarrow p^*; q \leq r$	KA-Lfp-L
$\mathsf{id} + p^*; p \equiv p^*$	KA-Unroll-R
$p + q; r \leq q \Rightarrow p; r^* \leq q$	KA-Lfp-R

Additional Boolean Algebra Axioms

$a + (b; c) \equiv (a + b); (a + c)$	BA-Plus-Dist
$a + \mathsf{id} \equiv \mathsf{id}$	BA-Plus-One
$a + \neg a \equiv \mathsf{id}$	BA-Excl-Mid
$a; b \equiv b; a$	BA-Seq-Comm
$a; \neg a \equiv \mathsf{drop}$	BA-Contra
$a; a \equiv a$	BA-Seq-Idem

Packet Algebra Axioms

$f \leftarrow n; f' \leftarrow n' \equiv f' \leftarrow n'; f \leftarrow n, \text{ if } f \neq f'$	PA-Mod-Mod-Comm
$f \leftarrow n; f' = n' \equiv f' = n'; f \leftarrow n, \text{ if } f \neq f'$	PA-Mod-Filter-Comm
$\mathsf{dup}; f = n \equiv f = n; \mathsf{dup}$	PA-Dup-Filter-Comm
$f \leftarrow n; f = n \equiv f \leftarrow n$	PA-Mod-Filter
$f = n; f \leftarrow n \equiv f = n$	PA-Filter-Mod
$f \leftarrow n; f \leftarrow n' \equiv f \leftarrow n'$	PA-Mod-Mod
$f = n; f = n' \equiv \mathsf{drop}, \text{ if } n \neq n'$	PA-Contra
$\sum_i f = i \equiv \mathsf{id}$	PA-Match-All

Figure 2.4: Equational axioms of NetKAT.

Producing the empty set models dropping the packet (and its history); producing a singleton set models modifying or forwarding the packet to a single location; and producing a set with multiple histories models modifying the packet in several ways or forwarding the packet to multiple locations. Note that policies only ever inspect or modify the first (current) packet in the history. This means implementations need not actually record histories—they are only needed for reasoning.

Figure 2.3 defines the denotational semantics of NetKAT. There is no separate definition for predicates—every predicate is a policy, and the semantics of $(;)$ and $(+)$ are the same whether they are composing policies or predicates. The syntactic distinction between policies and predicates arises solely to ensure that negation is only applied to a predicate, and not, for example, to a policy such as p^*. Formally, a predicate denotes a function that returns either the singleton $\{h\}$ or the empty set $\emptyset$ when applied to a history h. Hence, predicates behave like filters. A modification $(f \leftarrow n)$ denotes a function that returns a singleton history in which the field f of the current packet has been updated to n. Parallel composition $(p+q)$ denotes a function that produces the union of the sets generated by p and q, and sequential composition $(p;q)$ denotes the Kleisli composition $(\bullet)$ of the functions p and q, where the Kleisli composition of functions of type $\mathsf{H} \to \mathcal{P}(\mathsf{H})$ is defined as:

$$(f \bullet g)\ x \triangleq \bigcup \{g\ y \mid y \in f\ x\}\,.$$

Policy iteration p^* is interpreted as a union of semantic functions F_i of h, where each F_i is the Kleisli composition of function denoted by p i times. Finally, dup denotes a function that duplicates the current packet and adds it to the history. Since modification updates the packet at the head of the history, dup "freezes" the current state of the packet and makes it observable.

KAT-Invariant	If $a;p \equiv p;a$ then $a;p^* \equiv a;(p;a)^*$	Lemma 2.3.2 in [26]
KAT-Sliding	$p;(q;p)^* \equiv (p;q)^*;p$	Identity 19 in [26]
KAT-Denesting	$p^*;(q;p^*)^* \equiv (p+q)^*$	Identity 20 in [26]
KAT-Commute	If for all atomic x in q, $x;p \equiv p;x$ then $q;p \equiv p;q$	Corollary of Lemma 4.4 in [3]

Figure 2.5: KAT theorems.

2.2 Equational Theory

NetKAT, as its name suggests, is an extension of Kleene algebra with tests. Formally, a *Kleene algebra* (KA) is an algebraic structure $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$, where K is an idempotent semiring under $(+, \cdot, 0, 1)$, and $p^* \cdot q$ (respectively $q \cdot p^*$) is the least solution of the affine linear inequality $p \cdot r + q \leq r$ (respectively $r \cdot p + q \leq r$), where $p \leq q$ is an abbreviation for $p + q = q$. The axioms of KA are listed in Figure 2.4. The axioms are shown in the syntax NetKAT, which uses the more suggestive names ;, drop, and id for $\cdot$, 0, and 1, respectively.

A *Kleene algebra with tests* (KAT) is a two-sorted algebra

$$(K,\ B,\ +,\ \cdot,\ ^*,\ 0,\ 1,\ \neg)$$

with $\neg$ a unary operator defined only on B, such that

- $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ is a Kleene algebra,
- $(B,\ +,\ \cdot,\ \neg\ ,\ 0,\ 1)$ is a Boolean algebra, and
- $(B,\ +,\ \cdot,\ 0,\ 1)$ is a subalgebra of $(K,\ +,\ \cdot,\ 0,\ 1)$.

Elements of B and K are usually called *tests* and *actions* respectively; we will identify them with NetKAT predicates and policies. The axioms of Boolean algebra consist of the axioms of idempotent semirings (already listed as KA axioms) and the additional axioms listed in Figure 2.4.

It is easy to see that NetKAT has the required syntactic structure to be a KAT. However, the KAT axioms are not complete for the underlying NetKAT packet model. To establish completeness, we also need the packet algebra axioms listed in Figure 2.4. The first three axioms specify commutativity conditions. For example, PA-Mod-Mod-Comm states that assignments $\mathsf{src} \leftarrow X$ and $\mathsf{dst} \leftarrow Y$ can be applied in either order, as src and dst are different:

$$\mathsf{src} \leftarrow X; \mathsf{dst} \leftarrow Y \equiv \mathsf{dst} \leftarrow Y; \mathsf{src} \leftarrow X$$

Similarly, axiom PA-Mod-Filter-Comm states that the assignment $\mathsf{src} \leftarrow X$ and predicate $\mathsf{sw} = A$ can be applied in either order. The axiom PA-Dup-Filter-Comm states that every predicate commutes with dup. Interestingly, only this single axiom is needed to characterize dup in the equational theory. The next few axioms characterize modifications. The PA-Mod-Filter axiom states that modifying a field f to n and then filtering on packets with f equal to n is equivalent to the modification alone. Similarly, the axiom PA-Filter-Mod states that filtering on packets with field f equal to n and then modifying that field to n is equivalent to just the filter. PA-Mod-Mod states that only the last assignment in a sequence of assignments to the same f has any effect. The last two axioms characterize filters. The axiom PA-Contra states that a field cannot be equal to two different values at the same time. Finally, the axiom PA-Match-All states that the sum of filters on every possible value is equivalent to the identity. This implies packet values are drawn from a finite domain, such as fixed-width integers.

A simple example: access control. To illustrate the NetKAT equational theory, we prove a simple equivalence using the policies from the beginning of this chapter. Recall that the policy p_A filtered ssh packets on switch A while p_B filtered ssh packets on switch B. We prove that these programs are equivalent on ssh traffic going from

$$
\begin{aligned}
& \mathit{in}; (p_A; t)^*; p_A; \mathit{out} \\
\equiv\ & \{\ \text{definition } \mathit{in},\ \mathit{out}, \text{ and } p_A\ \} \\
& s_A; \mathrm{SSH}; ((s_A; \neg\mathrm{SSH}; p + s_B; p); t)^*; p_A; s_B \\
\equiv\ & \{\ \text{KAT-Invariant}\ \} \\
& s_A; \mathrm{SSH}; ((s_A; \neg\mathrm{SSH}; p + s_B; p); t; \mathrm{SSH})^*; p_A; s_B \\
\equiv\ & \{\ \text{KA-Seq-Dist-R}\ \} \\
& s_A; \mathrm{SSH}; (s_A; \neg\mathrm{SSH}; p; t; \mathrm{SSH} + s_B; p; t; \mathrm{SSH})^*; p_A; s_B \\
\equiv\ & \{\ \text{KAT-Commute}\ \} \\
& s_A; \mathrm{SSH}; (s_A; \neg\mathrm{SSH}; \mathrm{SSH}; p; t + s_B; p; t; \mathrm{SSH})^*; p_A; s_B \\
\equiv\ & \{\ \text{BA-Contra}\ \} \\
& s_A; \mathrm{SSH}; (s_A; \mathsf{drop}; p; t + s_B; p; t; \mathrm{SSH})^*; p_A; s_B \\
\equiv\ & \{\ \text{KA-Seq-Zero, KA-Zero-Seq,KA-Plus-Comm, KA-Plus-Zero}\ \} \\
& s_A; \mathrm{SSH}; (s_B; p; t; \mathrm{SSH})^*; p_A; s_B \\
\equiv\ & \{\ \text{KA-Unroll-L}\ \} \\
& s_A; \mathrm{SSH}; (\mathsf{id} + (s_B; p; t; \mathrm{SSH}); (s_B; p; t; \mathrm{SSH})^*); p_A; s_B \\
\equiv\ & \{\ \text{KA-Seq-Dist-L and KA-Seq-Dist-R}\ \} \\
& (s_A; \mathrm{SSH}; p_A; s_B)+ \\
& (s_A; \mathrm{SSH}; s_B; p; t; \mathrm{SSH}; (s_B; p; t; \mathrm{SSH})^*; p_A; s_B) \\
\equiv\ & \{\ \text{KAT-Commute}\ \} \\
& (s_A; s_B; \mathrm{SSH}; p_A)+ \\
& (s_A; s_B; \mathrm{SSH}; p; t; \mathrm{SSH}; (s_B; p; t; \mathrm{SSH})^*; p_A; s_B) \\
\equiv\ & \{\ \text{PA-Contra}\ \} \\
& (\mathsf{drop}; \mathrm{SSH}; p_A)+ \\
& (\mathsf{drop}; \mathrm{SSH}; p; t; \mathrm{SSH}; (s_B; p; t; \mathrm{SSH})^*; p_A; s_B) \\
\equiv\ & \{\ \text{KA-Zero-Seq, KA-Plus-Idem}\ \} \\
& \mathsf{drop} \\
\equiv\ & \{\ \text{KA-Seq-Zero, KA-Zero-Seq, KA-Plus-Idem}\ \} \\
& s_A; (p_B; t)^*; (\mathrm{SSH}; \mathsf{drop}; p + s_B; \mathsf{drop}; p; s_B) \\
\equiv\ & \{\ \text{PA-Contra and BA-Contra}\ \} \\
& s_A; (p_B; t)^*; (\mathrm{SSH}; s_A; s_B; p + s_B; \mathrm{SSH}; \neg\mathrm{SSH}; p; s_B) \\
\equiv\ & \{\ \text{KAT-Commute}\ \} \\
& s_A; (p_B; t)^*; (\mathrm{SSH}; s_A; p; s_B + \mathrm{SSH}; s_B; \neg\mathrm{SSH}; p; s_B) \\
\equiv\ & \{\ \text{KA-Seq-Dist-L and KA-Seq-Dist-R}\ \} \\
& s_A; (p_B; t)^*; \mathrm{SSH}; (s_A; p + s_B; \neg\mathrm{SSH}; p); s_B \\
\equiv\ & \{\ \text{KAT-Commute}\ \} \\
& s_A; \mathrm{SSH}; (p_B; t)^*; (s_A; p + s_B; \neg\mathrm{SSH}; p); s_B \\
\equiv\ & \{\ \text{definition } \mathit{in},\ p_B, \text{ and } \mathit{out}\ \} \\
& \mathit{in}; (p_B; t)^*; p_B; \mathit{out}
\end{aligned}
$$

Figure 2.6: Access control code motion proof.

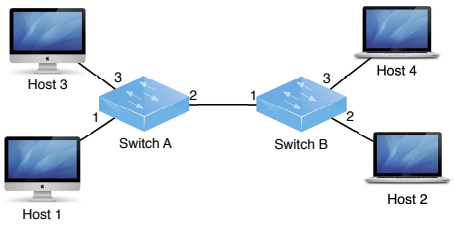

Figure 2.7: A simple network controlled by two parties.

left to right across the network topology shown in Figure 2.1. This can be seen as a simple form of code motion—moving the filter from switch A to switch B. We use the logical crossbar encoding with input and output predicates defined as follows:

$$in \triangleq (\mathsf{sw} = A; \mathsf{typ} = \text{SSH}) \qquad \text{and} \qquad out \triangleq (\mathsf{sw} = B)$$

The proof, given in Figure 2.6, is a straightforward calculation using the equational axioms and some standard KAT theorems, listed in Figure 2.5. The shaded term on each line indicates a term that will be changed in the next proof step. To lighten the notation we write s_A for $(\mathsf{sw} = A)$ and similarly for s_B, and SSH for $(\mathsf{typ} = \text{SSH})$.

2.3 Policy in Practice: Network Isolation

NetKAT's policy combinators help programmers construct rich network policies out of simpler parts. The most obvious combinator is union, which combines two smaller policies into one that, intuitively, provides the "union" of both behaviors. But naive use of union can lead to undesirable results due to interference between the packets generated by each sub-policy. This section explores how to use NetKAT's equational theory to reason about interference and presents a lightweight abstraction for enforcing isolation.

Example. Consider the network in Figure 2.7. Now, suppose the task of routing traffic between hosts 1 and 2 has been assigned to one programmer, while the task of routing traffic between hosts 3 and 4 has been assigned to another programmer. The first programmer might produce the following policy for switch B,

$$p_{B1} \triangleq \mathsf{sw} = B; (\mathsf{pt} = 1; \mathsf{pt} \leftarrow 2 + \mathsf{pt} = 2; \mathsf{pt} \leftarrow 1)$$

and the other programmer might produce a similar switch policy for B. This second policy differs from the first only by sending traffic from port 1 out port 3 rather than port 2:

$$p_{B2} \triangleq \mathsf{sw} = B; (\mathsf{pt} = 1; \mathsf{pt} \leftarrow 3 + \mathsf{pt} = 3; \mathsf{pt} \leftarrow 1)$$

Similar policies p_{A1} and p_{A2} define the behavior at switch A. Assume a topology assertion t that captures the topology of the network. By itself, the program

$$((p_{A1} + p_{B1}); t)^*$$

correctly sends traffic from host 1 to host 2. But when the second policy is added in,

$$(((p_{A1} + p_{B1}) + (p_{A2} + p_{B2})); t)^*$$

packets sent from host 1 will be copied to host 4 as well as host 2. In this instance, union actually produces *too many behaviors*. In the best case, sending additional packets to host 4 from host 1 leads to network congestion. In the worst case, it may violate the security policy for host 1. Either case demonstrates the need for better ways of composing forwarding policies.

Slices. A *network slice* [14] is a lightweight abstraction that facilitates modular construction of routing policies. Intuitively, a slice defines a piece of the network that

may be programmed independently. A slice comes with ingress (in) and egress (out) predicates, which define its boundaries, as well as an internal policy p. Packets that match in are injected into the slice. Once in a slice, packets stay in the slice and obey p until they match the predicate out, at which point they are ejected. We write slices as follows:

$$\{in\}\ w : (p)\ \{out\}$$

where in and out are the ingress and egress predicates and p defines the internal policy. Each slice also has a unique identifier w to differentiate it from other slices in a network program.[1]

It is easy to define slices by elaboration into NetKAT. We first create a new header field tag to record the slice to which the packet currently belongs.[2] In order for our elaboration to have the desired properties, however, the tag field must not be used elsewhere in the policy or in the ingress or egress predicates. We call a predicate *tag-free* if it commutes with any modification of the tag field, and a policy *tag-free* if it commutes with any test of the tag field.

Given tag-free predicates in, out and policy p, and a tag w_0 representing packets not in any slice, we can compile a slice into NetKAT as follows:

$$\begin{aligned}
&(\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0} \triangleq \\
&\quad \mathsf{let}\ pre = (\mathsf{tag} = w_0; in; \mathsf{tag} \leftarrow w + \mathsf{tag} = w)\ \mathsf{in} \\
&\quad \mathsf{let}\ post = (out; \mathsf{tag} \leftarrow w_0 + \neg out)\ \mathsf{in} \\
&\quad (pre; p; post)
\end{aligned}$$

Slice compilation wraps the slice policy with pre- and post-processing policies, pre and $post$. The pre policy tests whether a packet (i) is outside the slice (tagged with w_0) and matches the ingress predicate, in which case it is injected by tagging it with

[1] The unique identifier w may be defined by the compiler and need not actually appear in the surface syntax.

[2] In practice, the vlan field is often used to differentiate different classes of network traffic [56].

w, or (ii) has already been injected (already tagged with w). Once injected, packets are processed by p. If p emits a packet that matches the egress predicate *out*, then *post* strips the tag, restoring w_0. Otherwise, the packet remains in the slice and is left unmodified.

Isolation. A key property of slices is that once a packet enters a slice, it is processed solely by the slice policy until it is ejected, even across multiple hops in the topology.

Theorem 1. *Slice Composition*

For all tag*-free slice ingress and egress predicates* in, out*, identifiers* w*, policies* s, q*,* tag*-free policies* p*, and topologies* t*, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- *H0:* $w \neq w_0$,
- *H1:* $out; t; \mathsf{dup}; q \equiv \mathsf{drop}$,
- *H2:* $q; t; \mathsf{dup}; in \equiv \mathsf{drop}$,
- *H3:* q *drops* w*-tagged traffic, then*

$$((s + q); t; \mathsf{dup})^* \equiv (s; t; \mathsf{dup})^* + (q; t; \mathsf{dup})^*$$

Proof. The proof proceeds by induction on the structure of the policy p. This theorem appears as Theorem 7 in Appendix A, which presents the proof in full.

□

If the preconditions are met, including that the policy q neither processes traffic tagged with w nor sends traffic into or receives traffic from the boundary of s, then the theorem says that pushing a packet through a network executing s and q in parallel is the same as copying that packet and pushing it through two separate copies the network, one containing the slice and the other containing q. The proof of the theorem

is by equational reasoning and makes use of the KAT-Denesting lemma from Figure 2.5. (This theorem appears as Theorem 7 in Appendix A, which presents a full proof.)

An interesting corollary of the result above is that when the ingress slice boundary of s and the domain q do not overlap, and one restricts attention to traffic destined for the ingress of s, running s in parallel with q is equivalent to running s alone.

Corollary 1. *For all* tag*-free slice ingress and egress predicates in and out, identifiers* w*, policies* s, q*, and topologies* t*, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- *H0:* $w \neq w_0$,
- *H1:* $out; t; \mathsf{dup}; q \equiv \mathsf{drop}$,
- *H2:* $q; t; \mathsf{dup}; in \equiv \mathsf{drop}$,
- *H3:* $in; q \equiv \mathsf{drop}$*, then*

$$\begin{aligned} & in; \mathsf{tag} = w_0; ((s+q); t; \mathsf{dup})^* \\ \equiv\ & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* \end{aligned}$$

Proof. This corollary appears as Corollary 2 in Appendix A, which presents the proof in full.

□

Looking closely at Corollary 1, one can see a connection to traditional language-based information flow properties [44]. Think of s as defining the public, low-security data and q as defining the private, high security data. Under this interpretation, observable behavior of the network remains unchanged regardless of whether the high-security data (q) is present or replaced by some alternate high security data (q').

Example, redux. Slices provide a solution to the scenario described in the example at the beginning of the section. We can assign each programmer a unique slice with

boundaries that correspond to the locations of the end hosts under control of that slice. For instance, the first programmer's *in* and *out* predicates include the network access points for hosts 1 and 2, while the second programmer's *in* and *out* predicates include the network access points for hosts 3 and 4.

$$
\begin{aligned}
in_1 &= \mathsf{sw} = A; \mathsf{pt} = 1 + \mathsf{sw} = B; \mathsf{pt} = 2 \\
out_1 &= \mathsf{sw} = A; \mathsf{pt} = 1 + \mathsf{sw} = B; \mathsf{pt} = 2 \\
s_1 &= \{in_1\}\ w_1 : (p_{A1} + p_{B1})\ \{out_1\} \\
in_2 &= \mathsf{sw} = A; \mathsf{pt} = 3 + \mathsf{sw} = B; \mathsf{pt} = 3 \\
out_2 &= \mathsf{sw} = A; \mathsf{pt} = 3 + \mathsf{sw} = B; \mathsf{pt} = 3 \\
s_2 &= \{in_2\}\ w_2 : (p_{A2} + p_{B2})\ \{out_2\}
\end{aligned}
$$

The original difficulty with this example manifested as packet duplication when a packet was sent from host 1 to host 2. Corollary 1 shows that slices solve the problem: host 1 is connected to slice 1, and restricting the input to that of slice 1 implies that the behavior of the entire program is precisely that of slice 1 alone.

Toward pipelines and reconfiguration. The examples in this chapter make no assumptions about the shape of the underlying pipelines on switches in the network, other than to assume that each match and modification can be implemented in the hardware. Indeed, in this setting, we rely on each switch to implement the OpenFlow 1.0 protocol. The next chapter explores how to extend the switch-specific fragment of NetKAT to configure and interact with a more sophisticated pipeline abstraction.

Chapter 3

A Pipeline Configuration Language

The NetKAT language of the previous chapter was developed in the context of OpenFlow 1.0, with the intention of compiling NetKAT policies to configure switches that expose a single table of rules for packet processing. But modern switch architectures contain pipelines with many tables, and using a single table as a narrow waist—that is, compiling to a single table, and then distributing that table into a pipeline of tables—is inefficient, because it loses much of the structure of the original policy that can guide the deployment to a multi-table pipeline. In this chapter, we explore modifications to the NetKAT language for configuring and reasoning about multi-table switching pipelines. Intuitively, extending NetKAT is a natural approach; after all, a pipeline of tables strongly resembles a network of switches, at least from the perspective of packet forwarding.

We begin by restricting NetKAT in two ways. As our focus now lies within a single switch, we exclude statements that modify the logical "switch" field, such as $\mathsf{sw} \leftarrow A$, found in modeling topological links in the previous chapter. Similarly, the Kleene star construct, which characterizes unbounded iteration, has less relevance within the hard bounds of switching hardware, and so we remove it as well. We add to this base—which might be called *switch-local* NetKAT—primitives and combinators that

characterize switch pipelines, and we call the resulting language Concurrent NetCore, or CNC for short.

We introduce CNC through a series of examples, starting with user policies that define high-level packet processing, and then showing how CNC can model low-level switching hardware, before continuing with a formal presentation of the syntax and semantics (Section 3.1), detailed models of the Barefoot Networks' RMT and Intel's FlexPipe switch pipelines that are in production or under development today, and metatheoretical results (Section 3.3). Because CNC can model both ends of the spectrum, it can serve as a common intermediate language within an OpenFlow 2.0 compilation system. Chapter 4 illustrates this idea via algorithms that demonstrate how to transform our high-level user policies into a form that fits within the tables of the RMT pipeline.

CNC by example. Consider the illustration in Figure 3.1, which depicts several devices—a switch, a controller, a server and a DPI[1] box—as well as a link to "the internet." The switch has four ports (labelled 1, 2, 3, 4 in the picture) that connect it to the other devices and to the internet. Our goal is to write a policy for the switch that filters disallowed traffic, forwards permitted traffic, and diverts some traffic to the DPI box as part of a continually evolving intrusion-detection strategy.

Using the same notation as in the previous chapter, we can implement the following firewall w on the switch in Figure 3.1. It admits *ssh* or *http* traffic on port 1, but blocks all other traffic arriving on port 1. All traffic is allowed on ports other than 1. (Note that ; associates more tightly than +.)

$$w \quad \triangleq \quad \mathsf{in} = 1; (\mathsf{typ} = ssh + \mathsf{typ} = http) + \neg(\mathsf{in} = 1)$$

[1]DPI is *deep packet inspection*, a form of network security monitoring that inspects not just packet headers but their payloads as well.

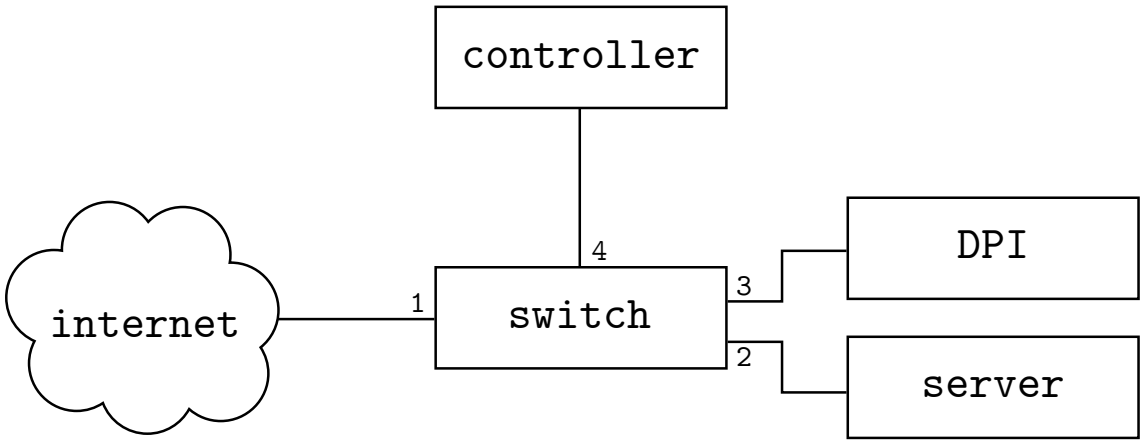

Figure 3.1: Another simple example network.

We can also define a simple, static forwarding policy that forwards packets from port 1 to port 2 and from port 2 to port 1.

$$r \quad \triangleq \quad \mathsf{in} = 1; \mathsf{out} \leftarrow 2 + \mathsf{in} = 2; \mathsf{out} \leftarrow 1$$

However, in order to serve as a configuration language for OpenFlow 2.0, we need a few more policy operators along with a simple type system for policies. First, the policies so far are completely static. They offer no room for populating new packet-processing rules at run time. To admit this kind of dynamic extension of static policies, we add *typed table variables*, which we write $(x : \tau)$. For example, we write $(x : (\{\mathsf{typ}, \mathsf{src}\}, \{\mathsf{out}\}))$ to indicate that the controller may later install new rules in place of x, and any such rules will only read from the typ and src header fields and write to the out field. The controller could use this table to dynamically install rules that forward selected subsets of packets to the DPI box for additional scrutiny. The typing information informs the switch of the kind of memory it needs to reserve for the table x (in this case, memory wide enough to be able to hold patterns capable of matching on both the typ and src fields).

A second key extension is concurrency, written $p_1 \,||\, p_2$. In order to reduce packet-processing latency within a switch, one may which to execute p_1 and p_2 concurrently on the *same* packet (rather than making copies). The latter is only legal provided there is no interference between subpolicies p_1 and p_2. In CNC, interference is prevented through the use of a simple type system. This type system prevents concurrent writes and ensures determinism of the overall packet-processing policy language.

As an example, consider the following policy p, which assembles each of the components described earlier. This policy checks for compliance with the firewall w while concurrently implementing a routing policy. The routing policy statically routes all packets to the server (this is the role of r) while dynamically selecting those packets to send to the DPI box (this is the role of x).

$$\begin{aligned} m &\triangleq (x : (\{\mathsf{typ}, \mathsf{src}\}, \{\mathsf{out}\})) \\ p &\triangleq w \,||\, (r + m) \end{aligned}$$

In essence, we have a form of *speculative execution* here. The policy $r + m$ is speculatively copying the packet and modifying it's out field while the firewall decides whether to drop it. If the firewall ultimately decides to drop the packet (*e.g.* because it is neither an SSH nor HTTP packet), then the results of routing and monitoring are thrown away. If the firewall allows the packet, then we have already computed how many copies of the packet are going out which ports. This kind of speculative execution is safe and deterministic when policies are well-typed.

If statements. As an aside, it is worth noting that the multicast combinator also serves as a form of disjunction. For example, consider the policy $a; p + \neg a; q$. The packet splits into two copies, but the predicates on the left- and right-hand sides of $+$ are disjoint—at least one copy will always be dropped. Hence, this particular form never actually produces multiple packet copies. It is useful to know syntactically that

no multicast happens—as we will see, it turns out that physical table stages contain sequences of nested if statements. We write (if a then p else q) for $(a; p + \neg a; q)$.

Modeling programmable hardware architectures. In addition to providing network administrators with a language for defining policies, our language of network policies aptly describes the hardware layout of switches' packet-processing pipelines. In this guise, table variables represent TCAM or SRAM tables, and combinators describe how these hardware tables are connected. The key benefit to devising a shared language for describing both user-level programs and hardware configurations is that we can define compilation as a semantics-preserving policy translation problem, and compiler correctness as a simple theorem about equivalence of input and output policies defined in a common language. Below, we demonstrate how to model key elements of the RMT [6] and FlexPipe [42] architectures. Both chips offer differently architectured fixed pipelines connecting reconfigurable tables.

In RMT (as well as in FlexPipe), multicast is treated specially: the act of copying and buffering multiple packets during a multicast while processing packets as quickly as they come in ("at line rate") is the most difficult element of chip design. The RMT multicast stage consists of a set of queues, one per output port. Earlier tables in the pipeline indicate the ports on which a packet should be multicast by setting bits in a metavariable bitmap we call out_i. The multicast stage consists of a sum, where each summand corresponds to a queue on a particular output port—when the ith out bit is set, the summand tags the packet with a unique identifier and sets its output port out to i accordingly.

$$\begin{aligned} \mathsf{multicast} \triangleq\ & (out_1 = 1; f_{\mathsf{tag}} \leftarrow v_1; \mathsf{out} \leftarrow 1) \\ +\ & (out_2 = 2; f_{\mathsf{tag}} \leftarrow v_2; \mathsf{out} \leftarrow 2) \\ +\ & \ldots \end{aligned}$$

In addition to the multicast processor, the RMT architecture provides thirty-two physical tables, which may be divided into sequences in the ingress and egress pipelines. The ingress pipeline processes an incoming packet first, before it is duplicated; any changes made at this point are copied into each duplicate. The egress pipeline, on the other hand, allows for individual handling—and modification of—each duplicated packet. The unique identifier (assigned in the multicast stage with $f_{\mathsf{tag}} \leftarrow v_k$) allows policy fragments in the egress pipeline to distinguish between packet copies. Overall, the RMT pipeline consists of the ingress pipeline, followed by the multicast stage, followed by the egress pipeline.

$$\begin{aligned} \mathsf{pipeline} \triangleq\ & (x_1 : \tau_1); \ldots; (x_k : \tau_k); \\ & \mathsf{multicast}; \\ & (x_{k+1} : \tau_{k+1}); \ldots; (x_{32} : \tau_{32}) \end{aligned}$$

The FlexPipe architecture makes use of concurrency by arranging its pipeline into a diamond shape. Each point of the diamond is built from two tables in sequence, with incoming packets first processed by the first pair, then concurrently by the next two pairs, and finally by the last pair. This built-in concurrency optimizes for common networking tasks, such as checking packets against an access control list while simultaneously calculating routing behavior.

$$\begin{aligned} \mathsf{pair}_i &\triangleq (x_{i,1} : \tau_{i,1}); (x_{i,2} : \tau_{i,2}) \\ \mathsf{diamond} &\triangleq \mathsf{pair}_1; (\mathsf{pair}_2 \mid\mid \mathsf{pair}_3); \mathsf{pair}_4 \end{aligned}$$

The FlexPipe multicast stage occurs after the diamond pipeline and, like the RMT multicast stage, relies on metadata set in the ingress pipeline to determine multicast. FlexPipe can make up to five copies ("mirrors") of the packet that can be independently modified, but each copy can be copied again to any output port, so long as no

further modifications are required.

$$\begin{aligned} \mathsf{multicast} &\triangleq \mathsf{mirror}; \mathsf{egress}; \mathsf{flood} \\ \mathsf{pipeline} &\triangleq \mathsf{diamond}; \mathsf{multicast} \end{aligned}$$

We present models of both RMT and FlexPipe (including mirror, egress and flood) in greater detail in Chapter 3.2.

3.1 Syntax and Semantics

We define the syntax of Concurrent NetCore in Figure 3.2. The language is broken into two levels: predicates and policies. Predicates, written with the metavariables a and b, simply filter packets without modifying or copying them. Policies, written with the metavariables p and q, can (concurrently) modify and duplicate packets. Every predicate is a policy—a read-only one. Both policies and predicates are interpreted using a *set semantics*, much like NetKAT [2]. Policies are interpreted as functions from sets of packets to sets of packets, while predicates have two interpretations: as functions from sets of packets to sets of packets, but also as Boolean propositions selecting a subset of packets. The latter interpretation, although subsumed by the former, highlights the difference between predicates and policies: predicates are policies that neither duplicate nor modify packets. A *packet*, written with the metavariable pk, is finite partial function from *fields* to *values*. We fix a set of fields F, from which we draw individual fields f. We will occasionally refer to sets of fields using the metavariables R and W when they denote sets of readable or writable fields, respectively. We do not have a concrete treatment for values $v \in \mathsf{Val}$, though Val must be finite and support a straightforward notion of equality. One could model both equality and TCAM-style wildcard matching, but for simplicity's sake, we stick with equality only.

As explained at the beginning of this chapter, the policies of Concurrent NetCore include the predicates as well as primitives for field modification, tables ($x : \tau$), sequential composition (;), parallel composition (+), and concurrency (||). One difference from our informal presentation earlier is that concurrent composition $p\ {}_{W_p}||_{W_q}\ q$ formally requires a pair of write sets W_p and W_q where W_p denotes the set of fields that p may write and W_q denotes the set of fields that q may write. Our operational semantics in Section 3.1 will in fact get *stuck* if p and q have a race condition, *e.g.*, have read/write dependencies. One NetKAT combinator that Concurrent NetCore does *not* include is Kleene star: in this work, we focus on *switch-local* policies meant for a single switch, rather than global policies describing the behavior of an entire network.

Table variables ($x : \tau$) are holes in a policy to be filled in by the controller with an initial policy, which the controller updates as the switch processes packets. The type $\tau = (\mathsf{R}, \mathsf{W})$ constrains the fields that the table may read from (R) and write to (W). For example, the rules that populate the table ($x : (\{\mathsf{src}, \mathsf{typ}\}, \{\mathsf{dst}\})$) can only ever read from the src and typ fields and can only ever write to the dst fields. In practice, this means that the controller can substitute in for x any policy matching its type (or with a more restrictive type).

A note on packet field dependences. Packet formats often have complex dependencies, e.g., if the Ethertype field is `0x800`, then the Ethernet header is followed by an IP protocol header. Switches handle attempts to match or modify a missing field at run time, although the specific behavior varies by target architecture. In the RMT chip, for instance, there is a valid bit indicating the presence (or absence) of each possible field. In OpenFlow 1.0 architectures, matching against a missing field always succeeds. In both cases, writing to a missing field is treated as a non-operation. Hence, we assume that each packet arriving at each switch contains fields $f_1, \ldots, f_k$,

Fields	$f \in \mathsf{F}, \mathsf{W}, \mathsf{R}$	$::=$	$f_1 \mid \cdots \mid f_k$
Packets	$pk \in \mathsf{PK}$	$::=$	$\mathsf{F} \rightharpoonup \mathsf{Val}$
Variables	$x, y \in \mathsf{Var}$		
Types	τ	$\in$	$\mathcal{P}(\mathsf{R}) \times \mathcal{P}(\mathsf{W})$
Predicates	a, b	$::=$	id *Identity (True)*
		$\mid$	drop *Drop (False)*
		$\mid$	$f = v$ *Match*
		$\mid$	$\neg a$ *Negation*
		$\mid$	$a + b$ *Disjunction*
		$\mid$	$a; b$ *Conjunction*
Policies	p, q	$::=$	a *Filter*
		$\mid$	$f \leftarrow v$ *Modification*
		$\mid$	$(x : \tau)$ *Table variable*
		$\mid$	$p + q$ *Parallel composition*
		$\mid$	$p; q$ *Sequential composition*
		$\mid$	$p \ {}_{W_p}\!\|\|_{W_q} \ q$ *Concurrent composition*
States	σ	$::=$	$\langle p, \delta \rangle$
Packet trees	δ	$::=$	$\langle \mathsf{PK}, \mathsf{W} \rangle$ *Leaves*
		$\mid$	$\langle \mathsf{par}\ \delta_1\ \delta_2 \rangle$ *Parallel processing*
		$\mid$	$\langle \mathsf{not}_{\mathsf{PK}}\ \delta \rangle$ *Pending negation*
		$\mid$	$\langle \mathsf{seq}\ \mathsf{PK} \rangle \delta$ *Sequential processing*
		$\mid$	$\langle \mathsf{con}_{\mathsf{W}}\ \delta_1\ \delta_2 \rangle$ *Concurrent processing*

Figure 3.2: Syntax of Concurrent NetCore.

although in practice the value associated with each field (which we treat abstractly) may be a distinguished "not present" value.

Small-step operational semantics. We give a small-step semantics for *closed* policies, *i.e.*, policies where table variables have been instantiated with concrete policies. By convention, we fix a set of fields $\mathsf{M} \subseteq \mathsf{F}$ as metadata fields; we assume that in all packets, these fields are initially set to 0. This definition is particularly convenient because it obviates the need for any explicit treatment of metadata in our policy language. In a complete compiler, a more careful account is necessary—there may only be a finite capacity on a target chip for metadata and virtual or computed fields. We can approximate these restrictions by fixing the size of M.

Just like the switches we are modeling, our policies actually work on packets one at a time: switches take an input packet and produce a (possibly empty) set of (potentially modified) output packets. As a technical convenience, our operational semantics generalizes this, modeling policies as taking a *set* of packets to a set of packets. Making this theoretically expedient choice—as we will show in Lemma 3—doesn't compromise our model's adequacy.

While other variants of NetCore/NetKAT use a denotational semantics, we use a completely new small-step, operational semantics in order to capture the interleavings of concurrent reads and writes of various fields of a packet. The interaction between (nested) concurrent processing of shared fields and packet-copying parallelism is quite intricate and hence deserves a faithful, fine-grained operational model. In Section 3.3, we define a type system that guarantees the strong normalization of all concurrent executions (Lemma 1 for normalization and Lemma 2 for confluence), and show that despite the concurrency, we can in fact use a NetKAT-esque set-theoretic denotational semantics to reason about policies at a higher level of abstraction if we so choose (Lemma 3 for the non-concurrent parts, Lemmas 4 and 5 for concurrency).

Using PK to range over sets of packets, we define the states σ for the small-step operational semantics $\sigma \to \sigma'$ in Figure 3.2. These states $\sigma = \langle p, \delta \rangle$ are pairs of a policy p and a packet tree δ. Packet trees represent the state of packet processing, keeping track of how the fields of each packet are split for concurrent processing and how packets are copied for union processing. Packet trees also store extra information needed for later steps, which is necessary in the case of negation.

The leaves of packet trees are of the form $\langle \mathsf{PK}, \mathsf{W} \rangle$, where PK is a set of packets and W is a set of fields indicating the current *write permission*. The write permission indicates which fields may be written; other fields present in the packets $pk \in \mathsf{PK}$ may be read but not written. Packet processing is done when we reach a terminal state, $\langle \mathsf{id}, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle$.

There are three kinds of packet tree branches. The packet tree branch $\langle \mathsf{par}\ \delta_1\ \delta_2 \rangle$ represents a parallel composition $p + q$ where p is operating on δ_1 and q is operating on δ_2. The packet tree branch $\langle \mathsf{not}_{\mathsf{PK}}\ \delta \rangle$ represents a negation $\neg a$ where a is running on δ—when a terminates with some set of packets PK', we will compute $\mathsf{PK} \setminus \mathsf{PK}'$, i.e., those packets *not* satisfying a. The packet tree branch $\langle \mathsf{seq}\ \delta \rangle$ a sequential composition $p; q$, where processing begins with p and, once p has reduced to id, continues with q. The packet tree branch $\langle \mathsf{con}_{\mathsf{W}}\ \delta_1\ \delta_2 \rangle$ represents a concurrent composition $p\ {}_{W_p}||_{W_q}\ q$ where p works on δ_1 with write permission W_p and q works on δ_2 with write permission W_q. We also store W, the write permission before concurrent processing, so we can restore it when p and q are done processing. We show how packet trees are used in more detail below.

We write $\sigma \to \sigma'$ to mean that the state σ performs a step of packet processing and transitions to the state σ'. Packet processing modifies the packets in a state and/or reduces the term. The step relation relies on several auxiliary operators on packets and packet sets. We read $pk[f := v]$ as, "update packet pk's f field with the value v;" and $pk \setminus \mathsf{F}$ as, "packet pk without the fields in F;" and $\mathsf{PK} \setminus \mathsf{F}$ as, "those packets in PK without the fields in F," which lifts $pk \setminus \mathsf{F}$ to sets of packets. Finally, we pronounce $\times$ as "cross product." Notice that $\mathsf{PK} \setminus \mathsf{F}$ only produces the empty set when PK is itself empty—if every packet $pk \in \mathsf{PK}$ has only fields in F, then $\mathsf{PK} \setminus \mathsf{F} = \{\bot\}$, the set containing the empty packet. Such a packet set is not entirely trivial, as there remains one policy decision to be made about such a set of packets: drop (using drop) or forward (using id)? On the other hand, $\emptyset \times \mathsf{PK} = \mathsf{PK} \times \emptyset = \emptyset$.

With these definitions in hand, we define the step relation in Figure 3.3, accompanied by auxiliary functions defined in Figure 3.4. The following invariants of evaluation and well-typed policies may be of use while reading through Figure 3.3.

Reduction relation

$\boxed{\sigma_1 \to \sigma_2}$

$$\frac{}{\langle \mathsf{drop}, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle \mathsf{id}, \langle \emptyset, \mathsf{W} \rangle \rangle} \textsc{Drop}$$

$$\frac{}{\langle f = v, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle \mathsf{id}, \langle \{pk \in \mathsf{PK} \mid pk(f) = v\}, \mathsf{W} \rangle \rangle} \textsc{Match}$$

$$\frac{f \in \mathsf{W}}{\langle f \leftarrow v, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle \mathsf{id}, \langle \{pk[f := v] \mid pk \in \mathsf{PK}\}, \mathsf{W} \rangle \rangle} \textsc{Modify}$$

$$\frac{}{\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle p; q, \langle \mathsf{seq}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle} \textsc{SeqEnter} \qquad \frac{\langle p, \delta \rangle \to \langle p', \delta' \rangle}{\langle p; q, \langle \mathsf{seq}\ \delta \rangle \rangle \to \langle p'; q, \langle \mathsf{seq}\ \delta' \rangle \rangle} \textsc{SeqL}$$

$$\frac{}{\langle \mathsf{id}; q, \langle \mathsf{seq}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle \to \langle q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle} \textsc{SeqR}$$

$$\frac{}{\langle p + q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle p + q, \langle \mathsf{par}\ \langle \mathsf{PK}, \mathsf{W} \rangle\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle} \textsc{ParEnter}$$

$$\frac{\langle p, \delta_p \rangle \to \langle p', \delta_p' \rangle}{\langle p + q, \langle \mathsf{par}\ \delta_p\ \delta_q \rangle \rangle \to \langle p' + q, \langle \mathsf{par}\ \delta_p'\ \delta_q \rangle \rangle} \textsc{ParL}$$

$$\frac{\langle q, \delta_q \rangle \to \langle q', \delta_q' \rangle}{\langle p + q, \langle \mathsf{par}\ \delta_p\ \delta_q \rangle \rangle \to \langle p + q', \langle \mathsf{par}\ \delta_p\ \delta_q' \rangle \rangle} \textsc{ParR}$$

$$\frac{}{\langle \mathsf{id} + \mathsf{id}, \langle \mathsf{par}\ \langle \mathsf{PK}_p, \mathsf{W} \rangle\ \langle \mathsf{PK}_q, \mathsf{W} \rangle \rangle \rangle \to \langle \mathsf{id}, \langle \mathsf{PK}_p \cup \mathsf{PK}_q, \mathsf{W} \rangle \rangle} \textsc{ParExit}$$

$$\frac{}{\langle \neg a, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle a, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle} \textsc{NotEnter}$$

$$\frac{\langle a, \delta \rangle \to \langle a', \delta' \rangle}{\langle a, \langle \mathsf{not}_{\mathsf{PK}}\ \delta \rangle \rangle \to \langle a', \langle \mathsf{not}_{\mathsf{PK}}\ \delta' \rangle \rangle} \textsc{NotInner}$$

$$\frac{}{\langle \mathsf{id}, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}_a, \mathsf{W} \rangle \rangle \rangle \to \langle \mathsf{id}, \langle \mathsf{PK} \setminus \mathsf{PK}_a, \mathsf{W} \rangle \rangle} \textsc{NotExit}$$

$$\frac{\mathsf{W}_p \cap \mathsf{W}_q = \emptyset \qquad \mathsf{W}_p \cup \mathsf{W}_q \subseteq \mathsf{W}}{\langle p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q, \langle \mathsf{con}_{\mathsf{W}}\ \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle\ \langle \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W}_q \rangle \rangle \rangle} \textsc{ConEnter}$$

$$\frac{\langle p, \delta_p \rangle \to \langle p', \delta_p' \rangle}{\langle p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q, \langle \mathsf{con}_{\mathsf{W}}\ \delta_p\ \delta_q \rangle \rangle \to \langle p'\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q, \langle \mathsf{con}_{\mathsf{W}}\ \delta_p'\ \delta_q \rangle \rangle} \textsc{ConL}$$

$$\frac{\langle q, \delta_q \rangle \to \langle q', \delta_q' \rangle}{\langle p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q, \langle \mathsf{con}_{\mathsf{W}}\ \delta_p\ \delta_q \rangle \rangle \to \langle p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q', \langle \mathsf{con}_{\mathsf{W}}\ \delta_p\ \delta_q' \rangle \rangle} \textsc{ConR}$$

$$\frac{}{\langle \mathsf{id}\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ \mathsf{id}, \langle \mathsf{con}_{\mathsf{W}}\ \langle \mathsf{PK}_p, \mathsf{W}_p \rangle\ \langle \mathsf{PK}_q, \mathsf{W}_q \rangle \rangle \rangle \to \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK}_q, \mathsf{W} \rangle \rangle} \textsc{ConExit}$$

Figure 3.3: Semantics of Concurrent NetCore.

Packet operations

$$pk[f := v] = \lambda f'.\begin{cases} v & f = f' \\ pk(f') & \text{otherwise} \end{cases}$$

$$pk \setminus \mathsf{F} = \lambda f.\begin{cases} \bot & f \in \mathsf{F} \\ pk(f) & \text{otherwise} \end{cases}$$

$$\mathsf{PK} \setminus \mathsf{F} = \{pk \setminus \mathsf{F} \mid pk \in \mathsf{PK}\}$$

$$pk_1 \times pk_2 = \lambda f.\begin{cases} pk_1(f) & \text{when } f \notin \mathsf{Dom}\,(pk_2) \\ pk_2(f) & \text{when } f \notin \mathsf{Dom}\,(pk_1) \\ pk_1(f) & \text{when } pk_1(f) = pk_2(f) \end{cases}$$

$$\mathsf{PK}_1 \times \mathsf{PK}_2 = \{\, pk_1 \times pk_2 |\, pk_1 \in \mathsf{PK}_1, pk_2 \in \mathsf{PK}_2\}$$

Figure 3.4: Auxiliary functions accompanying the semantics in Figure 3.3.

- Policy evaluation begins with a leaf $\langle \mathsf{PK}, \mathsf{W} \rangle$ and ends with a leaf $\langle \mathsf{PK}', \mathsf{W} \rangle$ with the same write permissions W.This invariant follows from normalization (Lemma 1).

- Policies may modify the values of existing fields within packets, but they cannot introduce new packets nor new fields—policies given the empty set of packets produce the empty set of packets.

The first few rules are straightforward. The (Drop) rule drops all its input packets, yielding $\emptyset$. In (Match), a match $\langle f = v, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle$ filters PK, producing those packets which have f set to v. In (Modify), a modification $\langle f \leftarrow v, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle$ updates packets with the new value v. Both (Match) and (Modify) can get stuck: the former if f is not defined for some packet, and the latter if the necessary write permission ($f \in \mathsf{W}$) is missing.

Sequential processing for $p; q$ is simpler: we mark that sequential processing has begin (SeqEnter), running p to completion (SeqL), and then running q on the resulting packets (SeqR). Intuitively, this is the correct behavior with regard to drop: if p drops all packets, then q will run on no packets, and will therefore produce no packets. One could imagine a shortcut rule to optimize execution that skips q altogether when p produces no packets, but we omit it for clarity.

The parallel composition $p + q$ is processed on $\langle \mathsf{PK}, \mathsf{W} \rangle$ in stages, like all of the remaining rules. First, (ParEnter) introduces new packet tree branch, $\langle \mathsf{par}\ \langle \mathsf{PK}, \mathsf{W} \rangle\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle$, duplicating the original packets: one copy for p and one for q. ParL and ParR step p and q in parallel, each modifying its local packet tree. When both p and q reach a terminal state, ParExit takes the union of their results. Note that ParExit produces the identity policy, id, in addition to combining the results of executing p and q, and we restore the initial write permissions W. As with NetKAT, $p + q$ has a set semantics, rather than bag semantics. If p and q produce an identical packet pk, only one copy of pk will appear in the result.

Negation $\neg a$, like parallel composition, uses a special packet tree branch (not)—in this case, to keep a copy of the original packets. Running $\neg a$ on PK, we first save a copy of PK in the packet tree $\langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle$ (NotEnter), preserving the write permissions. We then run a on the copied packets (NotInner). When a finishes with some PK_a, we look back at our original packets and return the saved packets *not* in PK_a (NotExit).

Concurrent composition is the most complicated of all our policies. To run the concurrent composition $p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q$ on packets PK with write permissions W, we first construct an appropriate packet tree (ConEnter). We split the packets based on two sets of fields: those written by p, W_p, and those written by q, W_q. We also store the original write permissions W—a technicality necessary for the metatheory, since in well typed programs $\mathsf{W} = \mathsf{W}_p \cup \mathsf{W}_q$ (see (Con) in the typing rules in Figure 3.10,

Section 3.3). The sub-policies p and q run on restricted views of PK, where each side can (a) read and write its own fields, and (b) read fields *not written by the other*. To achieve (a), we split W between the two. To achieve (b), we remove certain fields from each side: the sub-policy p will process $\mathsf{PK} \setminus \mathsf{W}_q$ under its own write permission W_p (ConL), while the sub-policy q will process $\mathsf{PK} \setminus \mathsf{W}_p$ under its own write permission W_q (ConR). Note that it is possible to write bad sets of fields for W_p and W_q in three ways: by overlapping, with W_p and W_q sharing fields (stuck in (ConEnter)); by dishonesty, where p tries to write to a field not in W_p (stuck later in (Modify)); and by mistake, with p reading from a field in W_q (stuck later in (Match)). While evaluation derivations of such erroneous programs will get stuck, our type system rules out such programs (Lemma 1). When both sides have terminated, we have sets of packets PK_P and PK_q, the result of p and q processing fragments of packets and concurrently writing to separate fields. We must then reconstruct a set of complete packets from these fragments. In (ConExit), the cross product operator $\times$ merges the writes from PK_p and PK_q. We take every possible pair of packets pk_p and pk_q from PK_p and PK_q and construct a packet with fields derived from those two packets. (It is this behavior that leads us to call it the 'cross product'.) In the merged packet pk, there are three ways to include a field:

1. We set $pk.f$ to be $pk_p.f$ when $f \notin \mathsf{Dom}\left(pk_q\right)$. That is, f is in W_p and may have been written by p.

2. We set $pk.f$ to be $pk_q.f$ when $f \notin \mathsf{Dom}\left(pk_p\right)$. Here, $f \in \mathsf{W}_q$, and q may have written to it.

3. We set $pk.f$ to $pk_p.f$, which is equal to $pk_q.f$. For a f to be found in both packets, it must be that $f \notin \mathsf{W}_p \cup \mathsf{W}_q$—that is, f was not written at all.

This accounts for each field in the new packet pk, but do we have the right number of packets? If p ran a parallel composition, it may have duplicated packets; if q ran drop,

it may have no packets at all. One guiding intuition is that well typed concurrent compositions $p \,||\, q$ should be equivalent to $p; q$ and $q; p$. (In fact, *all* interleavings of well typed concurrent compositions should be equivalent, but sequential composition already gives us a semantics for the 'one side first' strategy.) The metatheory in Section 3.3 is the ultimate argument, proving normalization and soundness (Lemma 1) and the properties of concurrent composition (Lemmas 4 and 5), but we can give some intuition by example:

- Suppose that $\mathsf{PK} = \{pk\}$ and that $p = f_1 \leftarrow v_1$ and $q = f_2 \leftarrow v_2$ update separate fields. In this case $\mathsf{PK}_p = \{(pk \setminus \{f_2\})[f_1 := v_1]\}$ and $\mathsf{PK}_q = \{(pk \setminus \{f_1\})[f_2 := v_2]\}$. Taking $\mathsf{PK}_p \times \mathsf{PK}_q$ yields a set containing a single packet pk', where $pk'(f_1) = v_1$ and $pk'(f_2) = v_2$, but $pk'(f) = pk(f)$ for all other—just as if we ran $p; q$ or $q; p$.

- Suppose that $p = \mathsf{id}$ and $q = \mathsf{drop}$. When we take $\mathsf{PK}_p \times \mathsf{PK}_q$, there are no packets at all in PK_q, and so there is no output. This is equivalent to running $\mathsf{id}; \mathsf{drop}$ or $\mathsf{drop}; \mathsf{id}$.

- Suppose that $p = f_1 \leftarrow v_1 + f_1 \leftarrow v_1'$ and $q = f_2 \leftarrow v_2$. Running $\{f_1\} \,||\, \{f_2\}\, pq$ on PK will yield

$$\begin{aligned}
\mathsf{PK}_p &= \{pk[f_1 := v_1] \mid pk \in \mathsf{PK} \setminus \{f_2\}\} \cup \\
&\quad\ \{pk[f_1 := v_1'] \mid pk \in \mathsf{PK} \setminus \{f_2\}\} \\
\mathsf{PK}_q &= \{pk[f_2 := v_2] \mid pk \in \mathsf{PK} \setminus \{f_1\}\} \\
\mathsf{PK}_p \times \mathsf{PK}_q &= \{pk[f_2 := v_2][f_1 := v_1] \mid pk \in \mathsf{PK}\} \cup \\
&\quad\ \{pk[f_2 := v_2][f_1 := v_1'] \mid pk \in \mathsf{PK}\}
\end{aligned}$$

  Which is the same as running $p; q$ or $q; p$.

We should note that $p \ {}_{\mathsf{W}_p}||_{\mathsf{W}_q} \ q$ is *not* the same as $p; q$ when W_p and W_q are incorrect, e.g., when p tries to write a field $f \notin \mathsf{W}_p$, or when q tries to read a field $f \in \mathsf{W}_p$. Sequential composition may succeed where concurrent composition gets stuck!

Limitations of concurrent composition. The annotations on the concurrency operator denote a strong form of concurrency: they bound the write operations that *may* occur, not those that will necessarily take place for a given packet. In other words, the operational semantics will get stuck processing some policies that do not, in fact, exhibit race conditions. Take, for example, the following contrived example.

$$(\mathsf{if}\ a\ \mathsf{then}\ f \leftarrow v_1\ \mathsf{else}\ \mathsf{id}) \,||\, (\mathsf{if}\ a\ \mathsf{then}\ \mathsf{id}\ \mathsf{else}\ f \leftarrow v_2)$$

Supposing that a does not match f, then no packet will ever be subject to both modifications: the if statements on each side of the concurrent composition branch on the same predicate, and so will apply the same branch, hence never both modifying the same field. But what annotation do we ascribe the concurrency operator? Both sides potentially modify f, and so the annotation must be ${}_{\{f\}}||_{\{f\}}$, which causes the operational semantics to get stuck (at ConEnter), and which the type system—presented in the next section—will reject.

The problem arises because the semantics demands the writable fields be split as soon as the concurrency operator is reached. So far, this seems a reasonable limitation, in that we have not yet encountered pipeline architectures that support two concurrent tables potentially writing to the same field. But these architectures are new, and the documentation specifying the lowest level of their behavior is not necessarily complete. In the future, it would be worth revisiting this point, both to again determine whether the behavior of these switches matches the model, and to extend the semantics to capture more fine-grained concurrent behavior.

Modeling the SDN controller. The operational semantics is defined on *closed* policies—that is, policies without table variables. At configuration time, the controller installs a (possibly open) policy on each switch, which tells the switch how to arrange its packet processing pipeline. Next, at population time, the controller will send messages to the switch instructing it to replace each abstract table variable with a concrete (closed) policy, after which packet processing proceeds as described by the operational semantics from Figure 3.3.

Definition 1. *Population-time updates and closing functions.*

$$\textbf{\textit{Table bindings}} \quad b \in \mathsf{Var} \rightharpoonup \mathsf{Policy}$$
$$\textbf{\textit{Closing functions}} \quad T_b \in \mathsf{Policy} \to \mathsf{Policy}$$

We model population-time updates as partial functions b that map table variables to closed policies—i.e. a closing substitution—which we call *table bindings*. The function $T_b\ (p)$ structurally recurses through a policy p, replacing each table variable x with $b(x)$. That is, the policy p is a configuration-time specification, and $T_b\ (p)$ is an instance of that specification populated according to the update function b. Population-time updates and closing functions will play a large role in Chapter 4, when we present a compilation algorithm for transforming a policy (and subsequent updates) to fit on a fixed target architecture.

3.2 Pipeline Models

In addition to serving programmers at a user level, our language of network policies can model the hardware layout of a switch's packet-processing pipeline. When we interpret Concurrent NetCore policies as pipelines, table variables represent TCAM or SRAM tables, and combinators describe how tables are connected.

Figure 3.5 presents a detailed model of the physical tables found in both the RMT and FlexPipe pipelines, while Figures 3.7 and 3.9 define the pipeline architectures themselves. Both architectures share some physical characteristics, including the physical layout of hardware tables. These physical tables are built from SRAM or TCAM memory and hold rules that match on packet header fields and, depending on the results of the match, modify the packet header. Each table has a fixed amount of memory, but it can be reconfigured, in the same way the height and width of a rectangle can vary as the area remains constant. The width of a table is determined by the number of bits it matches on from the packet header, and the height determines the number of rules it can hold. Hence, knowing in advance that the controller will only ever install rules that match on the src is valuable information, as it allows more rules to be installed. Although both chips support complex operations—such as adding and removing fields, arithmetic, checksums, and field encryption—we only model rewriting the value of header fields.

Physical tables are so-called match/action tables: the table comprises an ordered list of rules matching some fields on the header of a packet. The table selects a matching rule and executes its corresponding action. We model physical tables in the pipeline as table variables, so we must be careful that our compiler only substitutes in policies that look like rules in a match/action table. In an implementation of a compiler from Concurrent NetCore to a switch, we would have to actually translate the rule-like policies to the switch-specific rule population instructions. In our model and the proofs of correctness, we treat policies of the form

$$\mathsf{matches}; \mathsf{crossbar}; \mathsf{actions}$$

as rules (the translation to syntactically correct OpenFlow rules is straightforward enough at this point). The matches policy matches some fields and selects actions to

perform; the crossbar policy collects the actions selected, and then the actions policy runs them. (We elaborate on these phases below.) We believe that this is an adequate model, since it would not be hard to translate CNC policies in this form to rules for a particular switch. Our model requires that run-time updates to physical tables be of the form above; i.e., the binding $b(x : \tau)$ (Definition 1) has a rule-like tripartite structure.

Physical tables. Each variable mapped to a physical table by the binding $b(x : \tau)$ comprises three stages. The *match* stage is first. A single match (match_i) sets the metadata field act_i based on a subset of fields drawn from the packet header; here we write $\Pi_k\ f_k = v_k$ to stand for some sequence of matches $f_1 = v_1; \ldots; f_n = v_n$ for some length n. These fields implicitly determine the width of the match. The metadata field $\mathsf{act_i}$ holds an action identifier A_{fv} which encodes an action—update field f with value v—as a unique bit sequence. This is a stand-in for the slightly more structured action languages of the RMT and FlexPipe chips. The f index of action identifiers ranges over fields, grouping updates to the same field. For example, A_{out1} corresponds to updating the output port field out to 1. By construction, action selection is written to a metadata field act_i that is unique to that match, allowing for the match stage to execute multiple matches concurrently.

Once the act_i fields are set, the physical table has a *crossbar* that combines the metadata fields and selects the actions to execute—which we model with metadata fields do_{fv}, one for each A_{fv}; here we write $\Sigma_k\ f_k = v_k$ to stand for some disjunction of matches $f_1 = v_1 + \ldots + f_n = v_n$ for some number of summands n. Each field do_{fv} is consumed by an *action* stage, which runs the corresponding actions on the packet. Each action_f stage tests for actions denoting updates to field f, which allows actions to execute concurrently. Continuing the example, if A_{out1} was assigned to any act_i field in a match stage, then the crossbar stage will set $\mathsf{do}_{\mathsf{out}v}$ to 1, and the action stage will have a clause (if $\mathsf{do}_{\mathsf{out}v} = 1$ then out $\leftarrow v$ else $\ldots$).

Physical tables

$$
\begin{array}{rcl}
\mathsf{match}_i & ::= & \mathsf{id} \\
& | & \mathsf{if}\ \Pi_k\ f_k = v_k\ \mathsf{then}\ \mathsf{out}_i \leftarrow A_{fv}\ \mathsf{else}\ \mathsf{match}_i \\
\mathsf{matches} & = & \mathsf{match}_1\ ||\ \mathsf{match}_2\ ||\ \ldots \\
\mathsf{crossbar} & = & \mathsf{drop} \\
& | & \mathsf{if}\ \Sigma_k\ \mathsf{act}_k = A_{f_k v_k}\ \mathsf{then}\ \mathsf{do}_{fv} \leftarrow 1\ \mathsf{else}\ \mathsf{crossbar} \\
\mathsf{action}_j & = & \mathsf{id} \\
& | & \mathsf{if}\ \mathsf{do}_{f_j v_1} = 1\ \mathsf{then}\ f_j \leftarrow v_1\ \mathsf{else}\ \mathsf{action}_j \\
\mathsf{actions} & = & \mathsf{action}_1\ ||\ \mathsf{action}_2\ ||\ \ldots \\
\mathsf{physical} & = & x : \tau \\
T_b\ (\mathsf{physical}) & = & \mathsf{matches}; \mathsf{crossbar}; \mathsf{actions}
\end{array}
$$

Figure 3.5: A common model for physical tables.

As a larger example, suppose we would like to compile the routing and firewall policies ($r\ ||\ w$) from Figure 3.1 as a single physical table.

$$
\begin{array}{rcl}
r & = & \mathsf{in} = 1; \mathsf{out} \leftarrow 2 + \mathsf{in} = 2; \mathsf{out} \leftarrow 1 \\
w & = & \mathsf{in} = 1; (\mathsf{typ} = ssh + \mathsf{typ} = http) + \neg(\mathsf{in} = 1)
\end{array}
$$

First, let's fix four concrete action values—we'll say that a value of 11 means "modify the out field to 1" ($\mathsf{out} \leftarrow 1$); a value of 12 means "modify the out field to 2" ($\mathsf{out} \leftarrow 2$); a value of 31 means "do nothing" (id); and a value of 41 means "drop the packet"

(drop). We begin by defining two concurrent match stages, one each for r and w.

$$
\begin{aligned}
\mathsf{match}_r &= \mathsf{if}\ \mathsf{in} = 1\ \mathsf{then}\ \mathsf{act}_r \leftarrow 12 \\
&\quad\ \mathsf{else\ if}\ \mathsf{in} = 2\ \mathsf{then}\ \mathsf{act}_r \leftarrow 11 \\
&\quad\ \mathsf{else}\ \mathsf{act}_r \leftarrow 41 \\
\mathsf{match}_w &= \mathsf{if}\ \mathsf{in} = 1; \mathsf{typ} = ssh\ \mathsf{then}\ \mathsf{act}_w \leftarrow 31 \\
&\quad\ \mathsf{else\ if}\ \mathsf{in} = 1; \mathsf{typ} = http\ \mathsf{then}\ \mathsf{act}_w \leftarrow 31 \\
&\quad\ \mathsf{else\ if}\ \mathsf{in} = 1\ \mathsf{then}\ \mathsf{act}_w \leftarrow 41 \\
&\quad\ \mathsf{else}\ \mathsf{act}_w \leftarrow 31 \\
\mathsf{matches} &= \mathsf{match}_r \,||\, \mathsf{match}_w
\end{aligned}
$$

The match_r construct mirrors the structure of r, but rather than directly modifying the out field directly, it assigns an action identifier to the act_r metadata field. Encoding w is slightly more complex, thanks to the presence of disjunction (+) and negation. But it follows a similar pattern: In addition to converting w to a sequence of nested if statements, match_w assigns an action identifier to the act_w metadata field in place of taking an action directly. The $\mathsf{matches}$ stage is made up of match_r and match_w composed concurrently.

The $\mathsf{crossbar}$ stage collects the action values assigned in the $\mathsf{matches}$ stage in order to communicate them to the $\mathsf{actions}$ stage, where modifications to the packet header fields occur.

$$
\begin{aligned}
\mathsf{crossbar} &= \mathsf{if}\ \mathsf{act}_r = 11 + \mathsf{act}_w = 11\ \mathsf{then}\ \mathsf{do}_{11} \leftarrow 1 \\
&\quad\ \mathsf{else\ if}\ \mathsf{act}_r = 12 + \mathsf{act}_w = 12\ \mathsf{then}\ \mathsf{do}_{12} \leftarrow 1 \\
&\quad\ \mathsf{else\ if}\ \mathsf{act}_r = 31 + \mathsf{act}_w = 31\ \mathsf{then}\ \mathsf{do}_{31} \leftarrow 1 \\
&\quad\ \mathsf{else\ if}\ \mathsf{act}_r = 41 + \mathsf{act}_w = 41\ \mathsf{then}\ \mathsf{do}_{41} \leftarrow 1 \\
&\quad\ \mathsf{else}\ \mathsf{drop}
\end{aligned}
$$

The actions stage consumes the output of the crossbar in order to effect modifications to the header fields. Actions on the same field are grouped; in this case, modifications to the out field are handled by $\mathsf{action}_{\mathsf{out}}$. This allows each action group to be executed concurrently, because they operate on different fields by construction.

$$
\begin{aligned}
\mathsf{action}_{\mathsf{out}} &= \mathsf{if}\ \mathsf{do}_{11} = 1\ \mathsf{then}\ \mathsf{out} \leftarrow 1 \\
&\quad\ \mathsf{else\ if}\ \mathsf{do}_{12} = 1\ \mathsf{then}\ \mathsf{out} \leftarrow 2 \\
&\quad\ \mathsf{else}\ \mathsf{id} \\
\mathsf{action}_{\mathsf{id}} &= \mathsf{if}\ \mathsf{do}_{31} = 1\ \mathsf{then}\ \mathsf{id} \\
&\quad\ \mathsf{else}\ \mathsf{id} \\
\mathsf{action}_{\mathsf{drop}} &= \mathsf{if}\ \mathsf{do}_{41} = 1\ \mathsf{then}\ \mathsf{drop} \\
&\quad\ \mathsf{else}\ \mathsf{id} \\
\mathsf{actions} &= \mathsf{action}_{\mathsf{out}} \mid\mid \mathsf{action}_{\mathsf{id}} \mid\mid \mathsf{action}_{\mathsf{drop}}
\end{aligned}
$$

Separating tables into three stages may seem excessive, but suppose r also modified the typ field. In this case, $r \mid\mid w$ is no longer well typed (because r writes to typ while w reads from it), but we may still extract concurrency from $w; r$: By splitting reading and writing into separate phases, the match stage for applying the access control policy (match_w) can run concurrently with the match determining the output port (match_r) with little change from the example above. Concurrent processing like this is a key feature of both the RMT and FlexPipe architectures.

RMT. The RMT chip provides a thirty-two table pipeline divided into ingress and egress stages, which are separated by a multicast stage. Figure 3.6 depicts a visual representation of the pipeline, while Figure 3.7 defines the formal model. As a packet arrives, tables in the ingress pipeline act upon it before it reaches the multicast stage. To indicate that the packet should be duplicated, ingress tables mark a set of metadata fields corresponding to output ports on the switch. The multicast stage maintains a set of queues, one per output port. The chip enqueues a copy of the packet (really

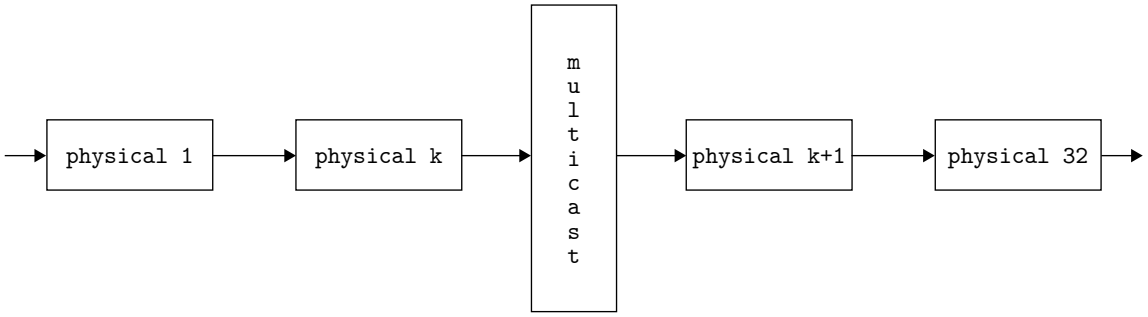

Figure 3.6: A depiction of the RMT pipeline.

RMT model

$$
\begin{array}{rcl}
\mathsf{multicast} & = & (out_1 = 1; f_{\mathsf{tag}} \leftarrow v_1; \mathsf{out} \leftarrow 1) \\
& + & (out_2 = 2; f_{\mathsf{tag}} \leftarrow v_2; \mathsf{out} \leftarrow 2) \\
& + & \ldots \\
\mathsf{pipeline} & = & \mathsf{physical}_1; \ldots; \mathsf{physical}_k; \mathsf{multicast}; \\
& & \mathsf{physical}_{k+1}; \ldots; \mathsf{physical}_{32} \\
& & \text{where } k \leq 16
\end{array}
$$

Figure 3.7: A linguistic model of the RMT pipeline.

a copy of the packet's header and a pointer to the packet's body) into those queues selected by the metadata, optionally marking each copy with a distinct tag. Finally, tables in the egress pipeline process each copy of the packet.

We model the multicast stage as the parallel composition of sequences of tests on header and metadata fields followed by the assignment of a unique value tag and an output port, where each summand corresponds to a queue in the RMT architecture. We model the ingress and egress pipelines as sequences of tables, where each of the thirty-two tables may be assigned to one pipeline or the other, but not both. The RMT architecture makes it possible to divide a single physical table into pieces and assign each piece to a different pipeline. We leave modeling this as future work.

FlexPipe. While physical tables have built-in concurrency within match and action stages, the FlexPipe architecture also makes use of concurrency between physical

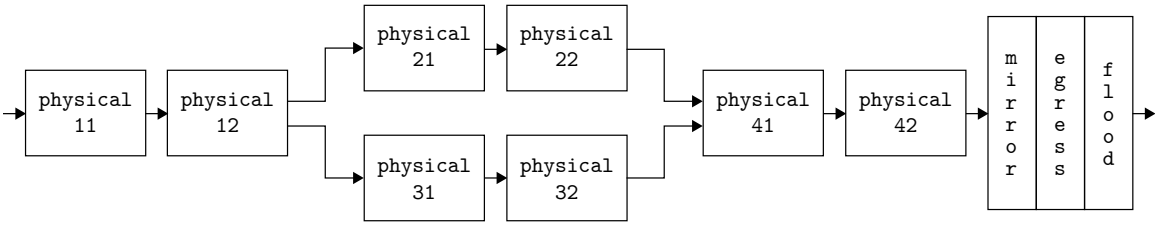

Figure 3.8: A depiction of the FlexPipe pipeline.

FlexPipe model

$$
\begin{aligned}
\mathsf{mirror} &= m = 0 + \textstyle\sum_i m_i = 1; m \leftarrow i \\
\mathsf{egress} &= \mathsf{if}\ f_{11} = v_{11}; \ldots; f_{1n} = v_{1n} \\
&\quad \mathsf{then}\ f'_{11} \leftarrow v'_{11}; \ldots; f'_{1n} \leftarrow v'_{1n} \\
&\quad \mathsf{else\ if}\ f_{21} = v_{21}; \ldots; f_{2m} = v_{2m} \\
&\quad \mathsf{then}\ f'_{21} \leftarrow v'_{21}; \ldots; f'_{2n} \leftarrow v'_{2n} \\
&\quad \mathsf{else}\ \ldots \\
\mathsf{flood} &= \textstyle\sum_i out_i = 1; \mathsf{out} \leftarrow i \\
\mathsf{pair} &= \mathsf{physical}_1; \mathsf{physical}_2 \\
\mathsf{diamond} &= \mathsf{pair}_1; (\mathsf{pair}_2 \parallel \mathsf{pair}_3); \mathsf{pair}_4 \\
\mathsf{pipeline} &= \mathsf{diamond}; \mathsf{mirror}; \mathsf{egress}; \mathsf{flood}
\end{aligned}
$$

Figure 3.9: A linguistic model of the Intel FlexPipe pipeline.

tables. Figure 3.8 depicts a visual representation of the pipeline, while Figure 3.9 defines the formal model.

The ingress pipeline is arranged in a diamond shape. Each point of the diamond is built from two tables in sequence, with incoming packets first processed by the first pair, then concurrently by the next two pairs, and finally by the last pair. This built-in concurrency is optimized for common networking tasks, such as checking packets against an access control list while simultaneously calculating routing behavior—as in our firewall example of Figure 3.1.

The FlexPipe architecture breaks multicast into two stages separated by a single egress stage. The mirror stage makes up to four additional copies of the packet. Each copy sets a unique identifier to a metadata field m and writes to a bitmap *out* corresponding to the ports on which this copy will eventually be emitted—this allows

for up to five potentially modified packets to be emitted from each port for each input packet. The egress stage matches on the metadata field m and various other fields to determine which modifications should be applied to the packet, and then applies those corresponding updates. Finally, the flood stage emits a copy of each mirrored packet on the ports set in its *out* bitmap.

3.3 Metatheory

The operational semantics of Section 3.1/Figure 3.3 defines the behavior of policies on packets. A number of things can cause the operational semantics to get stuck, which is how we model errors:

1. Unsubstituted variables—they have no corresponding rule.

2. Reads of non-existent fields—(Match) can't apply if there are packets $pk \in \mathsf{PK}$ such that $f \notin \mathsf{Dom}\,(pk)$, as might happen if ConEnter were to split packets incorrectly.

3. Writes to fields without write permission—(Modify) only allows writes to a field f if $f \in \mathsf{W}$.

4. Race conditions—concurrency splits the packet tree based on the write permissions of its subpolicies, and incorrect annotations can lead to stuckness via being unable to apply (ConEnter), which requires that $\mathsf{W}_p \cap \mathsf{W}_q = \emptyset$, or via getting stuck on (2) or (3) later in the evaluation due to the reduced fields and permissions each concurrent sub-policy runs with.

We define a type system in Figure 3.10, with the aim that well typed programs won't get stuck—a property we show in our proof of normalization, Lemma 1. First, we define entirely standard typing contexts, Γ. We will only run policies typed in the empty environment, i.e., with all of their tables filled in. Before offering typing rules

$$\Gamma ::= \cdot \mid \Gamma, (x : \tau)$$

$$(\mathsf{R}_1, \mathsf{W}_1) \cup (\mathsf{R}_2, \mathsf{W}_2) = ((\mathsf{R}_1 \setminus \mathsf{W}_2) \cup (\mathsf{R}_2 \setminus \mathsf{W}_1), \mathsf{W}_1 \cup \mathsf{W}_2)$$

$$\boxed{\vdash \tau} \qquad \boxed{\vdash \mathsf{PK} : \tau}$$

$$\frac{\mathsf{R}, \mathsf{W} \subseteq \mathsf{F} \qquad \mathsf{R} \cap \mathsf{W} = \emptyset}{\vdash (\mathsf{R}, \mathsf{W})} \qquad \frac{\forall pk \in \mathsf{PK}.\ \mathsf{R} \cup \mathsf{W} \subseteq \mathsf{Dom}\,(pk)}{\vdash \mathsf{PK} : (\mathsf{R}, \mathsf{W})}$$

$$\boxed{\Gamma \vdash p : \tau}$$

$$\frac{\vdash \tau}{\Gamma \vdash \mathsf{id} : \tau}\textsc{Id} \qquad \frac{\vdash \tau}{\Gamma \vdash \mathsf{drop} : \tau}\textsc{Drop} \qquad \frac{(x : \tau) \in \Gamma \qquad \vdash \tau}{\Gamma \vdash x : \tau}\textsc{Var}$$

$$\frac{\vdash (\mathsf{R}, \mathsf{W}) \qquad f \in \mathsf{R} \cup \mathsf{W}}{\Gamma \vdash f = v : (\mathsf{R}, \mathsf{W})}\textsc{Match} \qquad \frac{\vdash (\mathsf{R}, \mathsf{W}) \qquad f \in \mathsf{W}}{\Gamma \vdash f \leftarrow v : (\mathsf{R}, \mathsf{W})}\textsc{Modify}$$

$$\frac{\Gamma \vdash a : (\mathsf{R}, \emptyset)}{\Gamma \vdash \neg a : (\mathsf{R}, \mathsf{W})}\textsc{Not} \qquad \frac{\Gamma \vdash p : \tau_1 \qquad \Gamma \vdash q : \tau_2}{\Gamma \vdash p + q : (\tau_1 \cup \tau_2)}\textsc{Par} \qquad \frac{\Gamma \vdash p : \tau_1 \qquad \Gamma \vdash q : \tau_2}{\Gamma \vdash p; q : (\tau_1 \cup \tau_2)}\textsc{Seq}$$

$$\frac{\begin{array}{c}\Gamma \vdash p : (\mathsf{R}_p, \mathsf{W}_p) \\ \Gamma \vdash q : (\mathsf{R}_q, \mathsf{W}_q) \qquad \mathsf{W}_p \cap \mathsf{W}_q = \emptyset \qquad \mathsf{W}_p \cap \mathsf{R}_q = \emptyset \qquad \mathsf{R}_p \cap \mathsf{W}_q = \emptyset\end{array}}{\Gamma \vdash p \ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q : ((\mathsf{R}_p, \mathsf{W}_p) \cup (\mathsf{R}_q, \mathsf{W}_q))}\textsc{Con}$$

Figure 3.10: Typing rules for Concurrent NetCore.

for policies, we define well formedness of types and typing of packet sets. A type $\tau = (\mathsf{R}, \mathsf{W})$ is well formed if R and W are subsets of a globally fixed set of fields F and if $\mathsf{R} \cap \mathsf{W}$ is empty. A set of packets PK conforms to a type $\tau = (\mathsf{R}, \mathsf{W})$ if every packet $pk \in \mathsf{PK}$ has *at least* those fields in $\mathsf{R} \cup \mathsf{W}$.

The policies id and drop can both be typed at any well formed type, by (Id) and (Drop), respectively. Table variables $(x : \tau)$ are typed at their annotations, τ. The matching policy $f = v$ is well typed at τ when f is readable or writable (Match). Similarly, $f \leftarrow v$ is well typed at τ when f is writable in τ (Modify).

Negations $\neg a$ are well typed at $\tau = (\mathsf{R}, \mathsf{W})$ by (Not) when a is well typed at the read-only version of τ, i.e., $(\mathsf{R}, \emptyset)$. We restrict the type to being read-only to reflect

the fact that (a) only predicates can be negated, and (b) predicates never modify fields.

If p is well typed at τ_1 and q is well typed at τ_2, then their parallel composition $p + q$ is well typed at $\tau_1 \cup \tau_2$. Union on types is defined in Figure 3.3 as taking the highest privileges possible: the writable fields of $\tau_1 \cup \tau_2$ are those that were writable in either τ_1 or τ_2; the readable fields of the union are those fields that were readable in one or both types but weren't writable in either type. We give their sequential composition the same type.

Concurrent composition has the most complicated type—we must add (conservative) conditions to prevent races. Suppose $\Gamma \vdash p : (\mathsf{R}_p, \mathsf{W}_p)$ and $\Gamma \vdash q : (\mathsf{R}_q, \mathsf{W}_q)$. We require that:

- There are no write-write dependencies between p and q ($\mathsf{W}_p \cap \mathsf{W}_q = \emptyset$; a requirement of (ConEnter).
- There are no read-write or write-read dependencies between p and q ($\mathsf{W}_p \cap R_q = \emptyset$ and $\mathsf{R}_p \cap \mathsf{W}_q = \emptyset$). This guarantees that (Match) won't get stuck trying to read a field that isn't present.

If these conditions hold, then we say the concurrent composition is well typed: $\Gamma \vdash p \ {}_{W_p}||_{W_q} \ q : (\mathsf{R}_p, \mathsf{W}_p) \cup (\mathsf{R}_q, \mathsf{W}_q)$. Note that this means that the W stored in the con packet tree will be $\mathsf{W}_p \cup \mathsf{W}_q$, and well typed programs meet the $\mathsf{W}_p \cup \mathsf{W}_q \subseteq \mathsf{W}$ requirement of (ConEnter) exactly. These conditions are conservative—some concurrent compositions with overlapping reads and writes are race-free. We use this condition for a simple reason: switches make similar disjointness restrictions on concurrent tables.

Two metatheorems yield a strong result about our calculus: strong normalization. We first prove well typed policies are normalizing when run on well typed leaves

$\langle \mathsf{PK}, \mathsf{W} \rangle$—they reduce to the terminal state $\langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle \rangle$ with some other, well typed set of packets PK' and the same write permissions W.

Lemma 1 (Normalization)**.** *If*

$$\vdash \tau = (\mathsf{R}, \mathsf{W}) \text{ and } \vdash \mathsf{PK} : \tau \text{ and } \cdot \vdash p : \tau$$

then $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle \rangle$ *such that*

1. $\vdash \mathsf{PK}' : \tau$*, and*
2. $\mathsf{PK}' \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

Proof. By induction on the policy p, leaving τ general. The only difficulty is showing that (ConExit) can always successfully merge the results of well typed concurrency, which can be seen by a careful analysis of the cross product, using part (2) of the IH to show that fields not in the write permission W are "read-only". This lemma appears as Lemma 18 in Appendix B, which presents the proof in more detail.

□

Next we show that our calculus is confluent—even for ill typed terms. This result may be surprising at first, but observe that concurrency is the only potential hitch for confluence. A concurrent composition with an annotation that conflicts with the reads and writes of its sub-policies will get stuck before ever running (ConExit). Even ill typed programs will be confluent—they just might not be confluent at terminal states. We can imagine an alternative semantics, where concurrency really worked on shared state—in that formulation, only well typed programs would be confluent.

Lemma 2 (Confluence)**.** *If* $\sigma \to^* \sigma_1$ *and* $\sigma \to^* \sigma_2$ *then there exists* σ' *such that* $\sigma_1 \to^* \sigma'$ *and* $\sigma_2 \to^* \sigma'$.

Proof. By induction on the derivation of $\sigma \to^* \sigma_1$, proving the (stronger) single-step diamond property first. This lemma appears as Lemma 20 in Appendix B, which presents the proof in full.

□

Normalization and confluence yield *strong normalization*. Even though our small-step operational semantics is nondeterministic, well typed policies terminate deterministically. We can in fact do one better: our small-step semantics (without concurrency) coincides exactly with the denotational semantics of NetKAT [2], though we (a) do away with histories, and (b) make the quantification in the definition of sequencing explicit—as the union of q applied to each packet produced by p—rather than using Kleisli composition. Since our policies are 'switch-local', we omit Kleene star.

Lemma 3 (Adequacy)**.** *If* $\vdash \tau = (\mathsf{R}, \mathsf{W})$ *and* $\cdot \vdash p : \tau = (\mathsf{R}, \mathsf{W})$ *with no concurrency, then for all packets* $\vdash \mathsf{PK} : \tau$, $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ *and* $\mathsf{PK}' = \bigcup_{pk \in \mathsf{PK}} [\![p]\!]\, pk$*, where:*

$$
\begin{aligned}
[\![p]\!] &\in \mathsf{PK} \to \mathcal{P}(\mathsf{PK}) \\
[\![\mathsf{id}]\!]\, pk &\triangleq \{pk\} \\
[\![\mathsf{drop}]\!]\, pk &\triangleq \emptyset \\
[\![f = v]\!]\, pk &\triangleq \begin{cases} \{pk\} & pk(f) = v \\ \emptyset & \mathit{otherwise} \end{cases} \\
[\![f \leftarrow v]\!]\, pk &\triangleq \{pk[f := v]\} \\
[\![\neg a]\!]\, pk &\triangleq \{pk\} \setminus ([\![a]\!]\, pk) \\
[\![p + q]\!]\, pk &\triangleq [\![p]\!]\, pk \cup [\![q]\!]\, pk \\
[\![p; q]\!]\, pk &\triangleq \bigcup_{pk' \in [\![p]\!] pk} [\![q]\!]\, pk'
\end{aligned}
$$

Proof. By induction on $\cdot \vdash p : \tau$, noting that PK' always exists by the strong normalization result (Lemmas 1 and 2). This lemma appears as Lemma 22 in Appendix B, which presents the proof in full.

□

The set-based reasoning principles offered by the denotational semantics are quite powerful. We can in fact characterize the behavior of *well typed* concurrent compositions as:

$$\begin{aligned} [\![p \ {}_{W_p}||_{W_q} \ q]\!] &\triangleq [\![p; q]\!] \quad \text{(Lemma 5)} \\ &= [\![q; p]\!] \quad \text{(Lemma 4)} \end{aligned}$$

Lemma 4 (Concurrency commutes)**.** *If* $\vdash \mathsf{PK} : \tau$ *then*

$$\begin{aligned} &\vdash p \ \mathsf{w}_p||\mathsf{w}_q \ q : \tau \text{ and } \langle p \ \mathsf{w}_p||\mathsf{w}_q \ q, \mathsf{PK}\rangle \to^* \langle \mathsf{id}, \mathsf{PK}'\rangle \\ \iff &\vdash q \ \mathsf{w}_q||\mathsf{w}_p \ p : \tau \text{ and } \langle q \ \mathsf{w}_q||\mathsf{w}_p \ p, \mathsf{PK}\rangle \to^* \langle \mathsf{id}, \mathsf{PK}'\rangle. \end{aligned}$$

Proof. We reorder the congruence steps so that whenever we use ConL in one derivation, we use ConR in the other, and vice versa. Confluence (Lemma 2) proves the end results equal. □

Lemma 5 (Concurrency serializes)**.** *If* $\vdash p \ \mathsf{w}_p||\mathsf{w}_q \ q : (\mathsf{R}, \mathsf{W})$ *and* $\vdash \mathsf{PK} : \tau$ *then* $\langle p \ \mathsf{w}_p||\mathsf{w}_q \ q, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W}\rangle\rangle$ *iff* $\langle p; q, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W}\rangle\rangle$.

Proof. Rewriting derivations by confluence (Lemma 2) to run p using (ConL/SeqL) and then q (nesting in ConR under concurrency). We rely on auxiliary lemmas relating, for all p, p's behavior on $\mathsf{PK} \setminus \mathsf{W}_q$ and on PK (when $\mathsf{R}_p \cap \mathsf{W}_q = \mathsf{W}_p \cap W_q = \emptyset$). This lemma appears as Lemma 30 in Appendix B, which presents the proof in full. □

Together, Lemmas 3, 4, and 5 give us a means of using the NetKAT axioms, extended with axioms for concurrency, to reason about the transformation of CNC policies. The next chapter makes use of this technique to prove semantic preservation of several compilation algorithms.

Chapter 4

Compiling from High-level Policies to Low-level Pipelines

A major challenge for the designers of network programming languages is compiling their high-level abstractions to the low-level operations that network hardware supports. A pipeline-aware controller—that is, one that has visibility into and control over the packet-processing pipelines within the switches with which it connects—is faced with two compilation tasks. First, if the pipeline is reconfigurable, the controller must configure the pipeline to support *every* policy it might deploy during the lifetime of the network, or at least until each switch can be brought offline for reconfiguration. And second, it must decide how and where to install its rules at population time, when faced with a pipeline that may contain many tables with similar capabilities. Moreover, there may be many different switches in the network, each with its own pipeline and options for reconfiguration.

Both challenges cry out for an automated solution: While a network programmer could certainly write a different program for each type of switch in the network and reason manually about rule placement therein, the former task benefits immensely from a compiler, and the latter is better phrased as an optimization task.

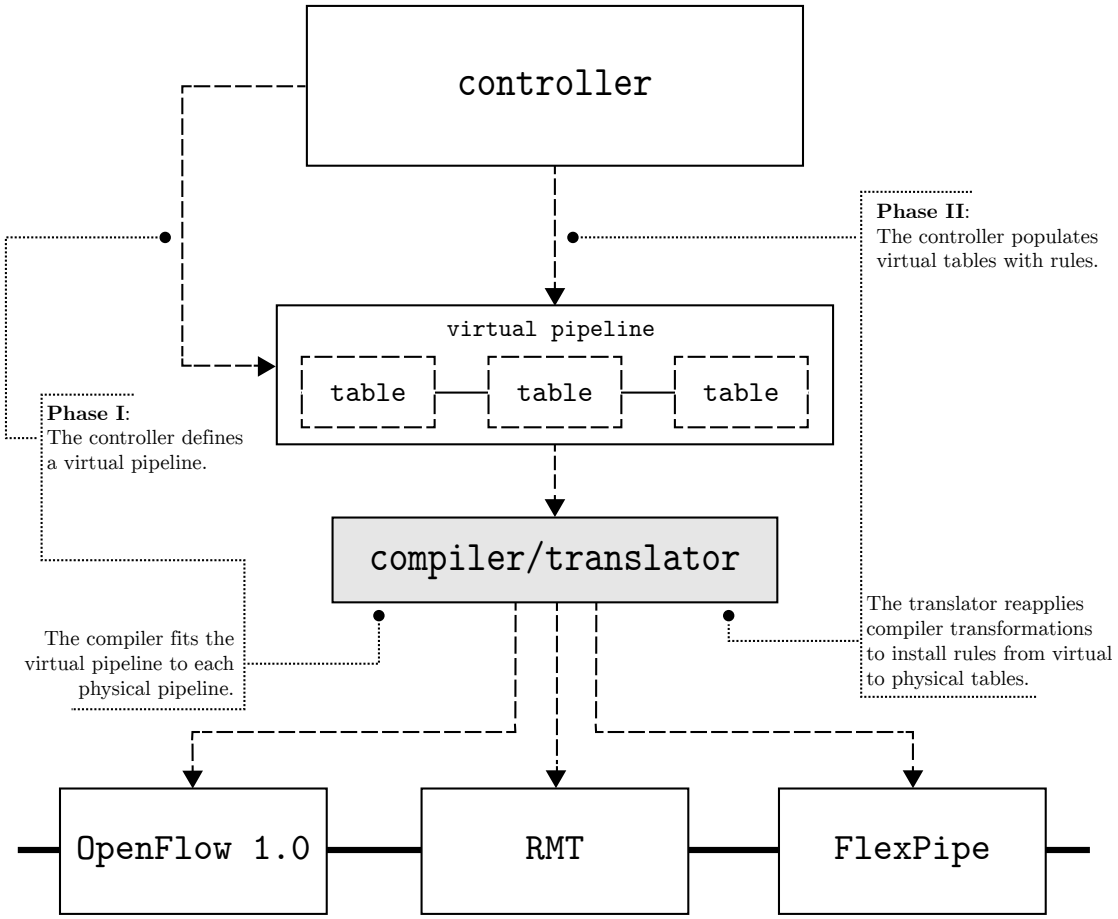

Figure 4.1: An example of the controller interacting with a virtual pipeline.

Virtual pipelines. In this chapter, we explore how the controller can represent its network resource requirements in the form of a *virtual pipeline*, and how a compilation framework can choose an optimal physical pipeline configuration for such a virtual pipeline and reconcile the differences between the virtual and physical pipelines through a set of compilation passes. Figure 4 illustrates the interaction between a controller, virtual pipeline, compiler, translator, and different physical pipelines. Notably, the virtual pipeline abstraction is naturally represented as a CNC policy: Just as CNC table variables and combinators can represent a physical pipeline, so too can they represent a virtual one. And so, the compilation process can be phrased as a translation of CNC policies, and the proof of compilation correctness—*i.e.* that

compilation is semantics preserving—becomes a proof of policy equivalence amenable to reasoning with the equational theory we developed in Chapter 3, Section 3.3.

Configuration-time pipeline compilation. Before the network begins transmitting data, the virtual pipeline must be compiled to fit within the physical pipeline. Compilation consists of several passes, each of which addresses a discrepancy between the expressivity of the high-level policy and the physical restrictions of the hardware model. In this section, we target the RMT architecture modeled in Section 3.2.

- **Multicast consolidation** transforms a policy with arbitrary occurrences of multicast (+) into a pipeline wherein multicast occurs at just a single stage.
- **If statement splitting** rewrites if statements with complex policies in the conditional branches into a sequence of two if statements, each with a complex policy in only one conditional branch.
- **Field extraction** moves modifications of a given field to an earlier stage of a pipeline.
- **Table fitting** partitions a pipeline into a sequence of tables, possibly combining multiple policy fragments into a single table, and introducing concurrent composition where possible.

At the end, the compiler produces a configuration for the physical pipeline as well as a population-time rule translator that reifies the placement and configuration decisions each pass makes.

Population-time rule translation. During compilation, some passes may move, modify, or break apart virtual tables to fit them into the physical pipeline. But at configuration time, tables are as yet unfilled; they are opaque variables that cannot themselves be modified. Rather, these operations must be applied to rules that are installed into the virtual tables at population time, in order to properly translate and

fit them into the appropriate physical tables. The compiler produces a *rule translator* to perform this function.

Single-table rule translation. The rule translator also performs a second task. Population-time rule updates may be largely unstructured, but the physical tables in switching hardware have a very fixed structure, as we saw in Section 3.2. The rule translator also transforms the rules to fit within the fixed structure of each physical table, which we call *single-table compilation*.

Hence, there are two challenges in deploying a CNC policy to a software-defined network: configuring the pipeline, and compiling population-time policies into the configured pipeline tables. We begin in Section 4.1 with the simpler exercise of compiling a NetKAT policy to a single table. Section 4.2 shows how to compile a virtual pipeline to the RMT pipeline model, and produce a population-time rule transformer that (a) reifies the compilation decisions in order to place rules populating virtual tables into the correct physical tables, and (b) applies single-table compilation.

A note on failure. The compilation of a well-typed virtual pipeline will always succeed, in that it will produce a physical pipeline that resembles the RMT pipeline and a transformer for handling population-time updates. But the compiled policy may require more resources than the RMT architecture can provide: either more tables (*i.e.* more physical memory for installing population-time rules) or more metadata fields. Similarly, population-time translation can also fail if the transformed update requires more resources than the physical architecture has available.

Each compilation stage takes an estimate of the resources required at population time; these estimates are used to reserve resources at compile time and potentially reject overly demanding programs. In practice, network programmers make such estimates by reasoning about the control logic that populates each table. For example, in a learning switch, the controller might install a new rule whenever a new host sends traffic through the switch. Hence, the size of the table can be bounded by the

maximum number of hosts allowed to join the network. This is as good as the current practice in many networks, where fixed table sizes implicitly bound network behavior. In the future, we hope to extend these compilation algorithms to accept weighted—rather than fixed—estimates, and to produce pipeline configurations that maximize the available rule space while respecting the relative weights of virtual tables.

4.1 Single-table Compilation

Before exploring end-to-end pipeline transformations, we examine the process by which an arbitrary switch-local NetKAT policy can be transformed to fit into a single OpenFlow 1.0 table. Other network programming languages have compilation algorithms with proofs of correctness [13, 37, 54, 10] that painstakingly relate their high-level semantics with the low-level semantics of OpenFlow rules. We take a different approach that allows us to exploit our equational theory: We define a syntactic restriction of NetKAT based on OpenFlow tables, which we call *OpenFlow Normal Form* (ONF) and then prove by induction (using purely syntactic arguments) that NetKAT can be normalized to ONF. We will use similar proof techniques to reason about pipeline transformations, and so this section serves as an introduction before moving on to more complex pipeline compilation. Notably, these same techniques can be applied to Concurrent NetCore population-time policies in order to install them into CNC tables, so long as the fields read and written by the policy match those supplied by the tables.

OpenFlow switches use a series of flow tables of rules to process packets. Because NetKAT supports richer expressions than OpenFlow, we must design compilation algorithms to "flatten" NetKAT policies into OpenFlow rule tables. The flattening algorithm proceeds by recursion on the structure of the policy, wherein each policy transformation is guided by the equational theory of Chapter 2. In fact, the algorithm

itself is presented as a proof of correctness of the single-table compilation procedure: Inductive steps in the proof can be interpreted as recursive calls in the algorithm, and applications of the equational axioms indicate transformations applied to the policy. The proof is lengthy and presented in full in Appendix C. A particularly challenging aspect lies in proving that sequential composition can be normalized: In an OpenFlow rule table (and thus in ONF) tests cannot be applied after actions, and so compilation must rewrite NetKAT terms—by carefully applying commutativity axioms—to commute the sequence of tests and actions, bringing tests to the front. This section highlights some of the important steps that enable the inductive normalization proof.

OpenFlow Normal Form (ONF). An OpenFlow 1.0-capable switch processes a packet using a table of prioritized rules [41]. In essence, each rule has a bit-pattern with wildcards to match packets and a list of actions to apply to packets that match the pattern. If a packet matches several rules, only the highest-priority rule is applied.

Instead of compiling NetKAT to rule tables directly, we define ONF, a syntactic subset of NetKAT that translates almost directly to rule tables. An ONF term is a cascade of if-then-else expressions, where the conditionals are a conjunction of positive literals, and the true branches are summations of actions, emitting one copy for each summand, or dropping the packet. Note that if-then-else is syntactic sugar.

$$\mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell \stackrel{\mathsf{def}}{=} (b; as) + (\neg b; \ell)$$

A policy p is in Openflow Normal Form ($p \in \mathsf{ONF}$) when it satisfies the grammar in Figure 4.2 and a few conditions. For example, consider the following policy.

$$\mathsf{typ} = ssh; \mathsf{out} \leftarrow 1 + \left(\begin{array}{l} \mathsf{if\ dstmac\ then\ 2\ else\ out} \leftarrow 2 \\ \mathsf{if\ dstmac\ then\ 3\ else\ out} \leftarrow 3 \\ \mathsf{drop} \end{array} \right)$$

ONF Action Sequence	a	::=	$\mathsf{id} \mid f \leftarrow n; a$
ONF Action Sum	as	::=	$\mathsf{drop} \mid a + as$
ONF Predicate	b	::=	$\mathsf{id} \mid f = n; b$
ONF	ℓ	::=	$as \mid \mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell$

Figure 4.2: OpenFlow Normal Form.

This policy monitors SSH traffic by copying it to port 1 and also switches local area traffic based on destination MAC address. The following is an equivalent policy in ONF.

$$\mathsf{if}\ \mathsf{typ} = ssh; \mathsf{dstmac} = 2\ \mathsf{then}\ (\mathsf{out} \leftarrow 1 + \mathsf{out} \leftarrow 2)\ \mathsf{else}$$

$$\mathsf{if}\ \mathsf{typ} = ssh; \mathsf{dstmac} = 3\ \mathsf{then}\ (\mathsf{out} \leftarrow 1 + \mathsf{out} \leftarrow 3)\ \mathsf{else}$$

$$\mathsf{drop}$$

There are a few side conditions on ONF policies. First, ONF excludes expressions that modify the switch field sw—this metadata field indicates the current switch processing the packet. And second, ONF excludes dup, which is used to record the hops a packet takes across the network for verifying reachability properties.

From NetKAT (and CNC) to ONF. Any NetKAT policy without the dup or Kleene star operators can be transformed into ONF.[1] We call NetKAT policies of this form "switch local." Notably, such policies coincide exactly with the fragment of CNC without concurrency or table variables, as might be supplied by the controller at population time.

Appendix C contains a detailed proof (Theorem 8) outlining how a switch-local NetKAT policy can be transformed into an equivalent ONF policy. The proof proceeds by induction on the structure of the policy, showing for each combinator how sub-policies in ONF can be combined to form a larger, equivalent policy in ONF. The proof can also be interpreted as a recursive algorithm to produce an equivalent ONF

[1] A NetKAT compiler will need a means of compiling Kleene star and specializing a network-wide policy to each switch, whereas the CNC language excludes Kleene star and already focuses on the pipeline within a single switch. Appendix C covers star elimination and switch localization procedures for NetKAT, offering a sense how CNC might be extended with similar constructs.

policy. The proof is lengthy and describes structural policy transformations in great detail, and the full presentation is relegated to the technical appendix.

As an example, consider the case for sequential composition: $p; q$. Suppose that, recursively, we have already computed equivalent policies in ONF for p and q; the task remains to combine them sequentially and form a single equivalent ONF policy.

$$p; q \equiv \left(\begin{array}{l} \mathsf{if}\ b_1\ \mathsf{then}\ as_1\ \mathsf{else} \\ \mathsf{if}\ b_2\ \mathsf{then}\ as_2\ \mathsf{else} \\ \ldots \end{array}\right); \left(\begin{array}{l} \mathsf{if}\ b_3\ \mathsf{then}\ as_3\ \mathsf{else} \\ \mathsf{if}\ b_4\ \mathsf{then}\ as_4\ \mathsf{else} \\ \ldots \end{array}\right)$$

This, in turn, can be accomplished by recursively joining the rows of each table: the first row of the left table with each row of the second, and so on, showing that the results themselves form a policy in ONF. Through the lens of the equational theory, this step can be viewed as desugaring the if statements into two large summations, one for each sub-policy.

$$\equiv \left(\begin{array}{l} b_1; as_1 + \\ \neg b_1; (b_2; as_2 + \neg b_2; \ldots) \end{array}\right); \left(\begin{array}{l} b_3; as_3 + \\ \neg b_3; (b_4; as_4 + \neg b_4; \ldots) \end{array}\right)$$

And distributing both sides using the Dist-L and Dist-R axioms of the equational theory.

$$\begin{aligned} \equiv\ & b_1; as_1; b_3; as_3 + \\ & b_1; as_1; \neg b_3; (b_4; as_4 + \neg b_4; \ldots) + \\ & \neg b_1; (b_2; as_2 + \neg b_2; \ldots); b_3; as_3 + \\ & \neg b_1; (b_2; as_2 + \neg b_2; \ldots); \neg b_3; (b_4; as_4 + \neg b_4; \ldots) \end{aligned}$$

Let's start with the first line. By comparing the predicates in b_3 with the modifications in as_1, we can compute a new set of predicates b_3' such that $as_1; b_3 \equiv b_3'; as_1$. For example, suppose $as_1 = \mathsf{out} \leftarrow 1$ and $b_3 = \mathsf{typ} = ssh; \mathsf{out} = 1$. In this case, we can see that $\mathsf{out} \leftarrow 1; \mathsf{typ} = ssh \equiv \mathsf{typ} = ssh; \mathsf{out} \leftarrow 1$ by the axiom PA-Mod-Filter-Comm.

And $\mathsf{out} \leftarrow 1; \mathsf{out} = 1 \equiv \mathsf{out} \leftarrow 1$ by PA-Mod-Filter, so we have $b_3' = \mathsf{typ} = ssh$, and

$$\mathsf{out} \leftarrow 1; \mathsf{typ} = ssh; \mathsf{out} = 1 \equiv \mathsf{typ} = ssh; \mathsf{out} \leftarrow 1.$$

Similar reasoning, along with an appeal to an induction hypothesis for resolving the smaller combinations on the last two lines, allows us to transform the entirety of this policy to ONF. The theorem (and algorithm) goes as follows.

Theorem 2 (Switch-local policies can be compiled to ONF)**.** *For all switch-local policies p, there exists a policy $p' \in$ ONF such that $p \equiv p'$.*

Proof. The proof proceeds by induction on the structure of the policy p. This theorem appears as Theorem 8 in Appendix C along with a complete proof.

□

From ONF to physical tables. The single-table compilation algorithm presented so far transforms switch-local NetKAT policies into OpenFlow 1.0 tables, *i.e.* sequences of if statements, one sequence per switch. Each if-statement sequence contains only positive conjunctions as predicates and sums of modifications in the true branches, where each summand represents one copy of the packet emitted by this rule. However, our model of physical tables from Section 3.2 is slightly more nuanced, consisting of separate match, crossbar, and action stages. We can still make use of the NetKAT compilation algorithm, but we add a final step transforming its output into our three-stage model.

First, we observe that in our compilation stack, single-table compilation always takes place in the transformer, at population time. Population-time rules in Concurrent NetCore do not contain table variables, and, as we saw in Chapter 3, the concurrent composition operator can always be replaced by sequential composition. What remains is a fragment of the CNC language that coincides with the switch-local fragment of NetKAT, making it a suitable target for this single-table compilation.

Physical tables

$$
\begin{array}{rcl}
\mathsf{match}_i & = & \mathsf{if}\ f_{11} = v_{11}; \ldots; f_{1n} = v_{1n} \\
& & \mathsf{then}\ \mathsf{act}_i \leftarrow A_{j1} \\
& & \mathsf{else\ if}\ f_{21} = v_{21}; \ldots; f_{2m} = v_{2m} \\
& & \mathsf{then}\ \mathsf{act}_i \leftarrow A_{k2} \\
& & \mathsf{else}\ \ldots \\
\mathsf{matches} & = & \mathsf{match}_1 \,||\, \mathsf{match}_2 \,||\, \ldots \\
\mathsf{crossbar} & = & \mathsf{if}\ \mathsf{act}_1 = A_{11} + \mathsf{act}_2 = A_{11} + \ldots \\
& & \mathsf{then}\ \mathsf{do}_{A_{11}} \leftarrow 1 \\
& & \mathsf{else\ if}\ \mathsf{act}_1 = A_{21} + \mathsf{act}_2 = A_{21} + \ldots \\
& & \mathsf{then}\ \mathsf{do}_{A_{21}} \leftarrow 1 \\
& & \mathsf{else}\ \ldots \\
\mathsf{action}_j & = & \mathsf{if}\ \mathsf{do}_{A_{j1}} = 1\ \mathsf{then}\ \text{ perform } A_{j1}\text{'s writes} \\
& & \mathsf{else\ if}\ \mathsf{do}_{A_{j2}} = 1\ \mathsf{then}\ \text{ perform } A_{j2}\text{'s writes} \\
& & \ldots \\
\mathsf{actions} & = & \mathsf{action}_1 \,||\, \mathsf{action}_2 \,||\, \ldots \\
\mathsf{physical} & = & x : \tau \\
T_b\ (\mathsf{physical}) & = & \mathsf{matches}; \mathsf{crossbar}; \mathsf{actions}
\end{array}
$$

Figure 4.3: A common model for physical tables, reproduced from Figure 3.5 in Section 3.2 for convenience.

Next, we observe that single-table compilation always takes place after multicast consolidation removes packet duplication (presented in Section 4.2). Hence, by construction, we will never compile a policy containing multicast, and the true branch in the single-table compilation output will contain a sequence of modifications, but never a sum of sequences. This fragment of ONF can be modeled as follows, where we write $\Pi_i p_i$ for $p_1; p_2; \ldots; p_n$.

$$ons ::= \mathsf{drop} \mid \mathsf{if}\ \Pi_i f_i = v_i\ \mathsf{then}\ \Pi_j f_j \leftarrow v_j\ \mathsf{else}\ ons$$

Recall from Figure 3.5 in Section 3.2 that we model a single physical table as a sequence of a match stage, a crossbar, and an action stage. For convenient reference, the model is reproduced here in Figure 4.3.

Definition 2 (Restricted ONF to physical tables)**.**

$$
\begin{array}{lcl}
\mathsf{crossbar\ drop} & = & (\mathsf{drop}, \mathsf{drop}, \mathsf{drop}) \\
\mathsf{crossbar\ if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ ons & = & \mathsf{let}\ m, c, t = \mathsf{crossbar}\ ons\ \mathsf{in} \\
 & & \mathsf{let}\ v = \mathit{fresh\ value}\ \mathsf{in} \\
 & & (\mathsf{if}\ a\ \mathsf{then}\ \mathsf{act} \leftarrow v\ \mathsf{else}\ m, \\
 & & \mathsf{if}\ \mathsf{act} = v\ \mathsf{then}\ \mathsf{do}_v \leftarrow 1\ \mathsf{else}\ c, \\
 & & \mathsf{if}\ \mathsf{do}_v = 1\ \mathsf{then}\ p\ \mathsf{else}\ t)
\end{array}
$$

The crossbar function transforms an *ons* table into match, crossbar, and action stages recursively. Suppose we examine the first row of an *ons* table: if a then p else *ons*. The predicate a helps form the match stage: If a holds, then the match stage updates the act field with a unique value v. The crossbar stage is extended to set do_v if act was set to v, tying the match to the appropriate action. Finally, the action stage applies the original actions p if do_v was set. Recursively applying *crossbar* yields match, crossbar, and action stages (m, c, t) for the remaining rows. Note that this is somewhat inefficient. A more sophisticated algorithm would map unique action sequences in *ons* to unique values and reuse those values where the action sequences are repeated.

This extension has also been proved to be semantics preserving, in Section C.1.5.

Lemma 6 (Single-table compilation from ONF to physical tables is semantics preserving)**.** *For all ONF tables ons without packet duplication, let $m, c,$ and t be match, crossbar, and action stages such that $m; c; t =$ crossbar ons, and let fs be the fresh metadata fields introduced by crossbar, with $z = \Pi_{f \in fs} f \leftarrow 0$ zeroing these fields. The following equivalence holds.*

$$z; m; c; t; \equiv; z; ons; z$$

Proof. The proof proceeds by induction on the structure of the *ons* table. This lemma appears in Appendix C as Lemma 54 along with a complete proof.

□

4.2 Pipeline Compilation

Pipeline compilation consists of several passes, each of which addresses a discrepancy between the expressivity of the high-level pipeline policy and the physical restrictions of the hardware model. In this section, we explore four compiler passes targeting the RMT architecture, as modeled in Section 3.2.

- **Multicast consolidation.** The RMT pipeline only supports packet duplication at one specific stage in the pipeline, whereas a virtual pipeline might use packet duplication throughout. *Multicast consolidation* transforms a policy with arbitrary occurrences of packet duplication (+) into a pipeline wherein duplication occurs at just a single stage.

- **If statement splitting.** As we saw in Section 3.2, the RMT pipeline is composed of a sequence of tables. In order to more efficiently spread one if statement—as might be produced by multicast consolidation—across multiple tables, *if statement splitting* transforms an if statement into two simpler if statements, composed sequentially.

- **Field extraction.** The RMT pipeline also requires that the output port for each packet be set during packet duplication. *Field extraction* moves modifications of a given field—such as the `out` field, which indicates the output port—to an earlier stage of a pipeline.

- **Table fitting.** The previous passes produce a pipeline of tables, composed sequentially, that must be mapped to the sequence of physical tables in the RMT

pipeline. *Table fitting* determines the best mapping, replacing sequential composition with concurrent composition where possible to improve performance.

The restrictions present in the RMT architecture are not uncommon; many pipelines (including FlexPipe) also disallow arbitrary packet duplication, compose physical tables sequentially, at least in part, and limit field modification to particular places in the pipeline. Hence, we believe these passes will be useful in targeting other architectures in the future.

Each pass takes a well-typed policy as input and produces an equivalent, refactored policy as well as a *binding transformer* as output.

Definition 3 (Binding transformer)**.** *A binding transformer θ is an operator on table bindings b.*

$$\theta \quad \in \quad (\mathsf{Var} \rightharpoonup \mathsf{Policy}) \to \mathsf{Var} \rightharpoonup \mathsf{Policy}$$

Binding transformers play the role of the "generated rule translator" from Figure 1.2. In other words, during the switch population phase, the controller will issue table bindings b—essentially, closing substitutions, see Definition 1 in Section 3.1—in terms of the (pre-compilation) virtual pipeline. The job of the binding transformer θ is to transform table bindings so they can be sensibly applied to the (post-compilation) physical pipeline configured on the switch.

Not every table binding constitutes a valid update. Definition 4 outlines the criteria under which a table binding is well formed with respect to a policy it updates.

Definition 4 (Well-formed table bindings)**.** *A table binding b is well formed with respect to a policy p, written $p \vdash b$ wf, if for all table variables $(x : \tau)$ in p, $\vdash (T_b\ x : \tau) : \tau$.*

A table binding is well formed if the policy substituted for each table is well typed at the same type annotating the table, and the substituted policy is closed (*i.e.* does not contain table variables).

4.2.1 Multicast Consolidation

There are two important differences between the kind of multicast (*i.e.* packet duplication) that Concurrent NetCore offers and the kind supported by the RMT pipeline described in Section 3.2. First, multicast may not occur arbitrarily in the RMT pipeline; rather, there is a fixed multicast stage sandwiched between two pipelines. Second, the multicast stage must know the destination output port of each packet copy *at the time the packet is copied. Multicast consolidation* rewrites a high-level policy into a form with a distinct multicast stage. The next section describes how *field extraction* extracts potential modifications to a given field from a sub-policy—which we use to isolate writes to the output port to the multicast stage.

Consider the following policy as a small, running example we will explore to showcase each of the compilation stages.

$$
\begin{array}{lcl}
p & \triangleq & \mathsf{typ} = ssh; \mathsf{out} \leftarrow 1 + \left(\begin{array}{l}
\mathsf{if\ ethtyp} = arp\ \mathsf{then} \\
\quad \mathsf{if\ dstmac} = 2\ \mathsf{then\ out} \leftarrow 2\ \mathsf{else} \\
\quad \mathsf{if\ dstmac} = 3\ \mathsf{then\ out} \leftarrow 3\ \mathsf{else} \\
\quad \mathsf{else\ drop} \\
\mathsf{else} \\
\quad x : (\{\mathsf{dstip}\}, \{\mathsf{out}\}) + y : (\{\mathsf{typ}\}, \{\mathsf{out}\})
\end{array} \right)
\end{array}
$$

The policy (*a.k.a.* virtual pipeline) p has three parts. First, the left sub-policy of the union ($\mathsf{typ} = ssh; \mathsf{out} \leftarrow 1$) statically copies all SSH traffic to port 1, perhaps for security monitoring. We say this fragment is *fixed* because it retains this behavior regardless of any population time updates. The *true* branch of the if statement handles local area switching—for the sake of simplicity, two local hosts have been hard-coded at destination MAC addresses 2 and 3. Finally, the *false* branch contains two table variables joined by union; the first governs destination IP routing, to be filled in at population time, and the second allows more fine-tuned monitoring at

population time. If we assume that neither x nor y will make any extra copies of the packet, then this policy will produce at most three new packets: one emitted from port 1 if the incoming packet is an SSH packet, another dictated by the table x, and a third from table y.

Multicast consolidation rewrites p into two stages: the *consolidation* stage makes three copies of the packet and sets a set of fresh metadata fields, the combination of which is unique to each packet, differentiating the three copies.

$$p_c \triangleq \mathsf{typ} = ssh + \mathsf{ethtyp} = arp + \neg\mathsf{ethtyp} = arp; f_1 \leftarrow 1 + \neg\mathsf{ethtyp} = arp; f_1 \leftarrow 1; f_2 \leftarrow 1$$

Each summand corresponds to a potential packet copy, but the filters control which duplicates are made for each specific input packet. For example, only two copies are made for ARP packets, while non-ARP packets are copied three times, which is in line with the behavior of the original policy. By convention, the metadata fields are initialized to zero; hence, $\mathsf{typ} = ssh$ indicates that both f_1 and f_2 are set to 0. Next, the *egress* stage replaces the original occurrences of multicast in p with a sequence of tests on the new metadata fields.

$$p_e \triangleq \left(\begin{array}{l}\mathsf{if}\ f_1 = 0\ \mathsf{then}\\ \quad \mathsf{out} \leftarrow 1\\ \mathsf{else\ id}\end{array}\right); \left(\begin{array}{l}\mathsf{if}\ f_1 = 1\ \mathsf{then}\\ \quad \mathsf{if}\ \mathsf{ethtyp} = arp\ \mathsf{then}\\ \quad\quad \mathsf{if\ dstmac} = 2\ \mathsf{then\ out} \leftarrow 2\ \mathsf{else}\\ \quad\quad \mathsf{if\ dstmac} = 3\ \mathsf{then\ out} \leftarrow 3\ \mathsf{else}\\ \quad\quad \mathsf{else\ drop}\\ \quad \mathsf{else}\\ \quad\quad \left(\begin{array}{l}\mathsf{if}\ f_2 = 0\ \mathsf{then}\\ x : (\{\mathsf{dstip}\}, \{\mathsf{out}\})\\ \mathsf{else\ id}\end{array}\right); \left(\begin{array}{l}\mathsf{if}\ f_2 = 1\ \mathsf{then}\\ y : (\{\mathsf{typ}\}, \{\mathsf{out}\})\\ \mathsf{else\ id}\end{array}\right)\\ \mathsf{else\ id}\end{array}\right)$$

The consolidation and egress stages are composed sequentially. Hence, $p_c; p_e$ acts equivalently to p, producing at most three packets: one processed by $\mathsf{out} \leftarrow 1$, another by $x : (\{\mathsf{dstip}\}, \{\mathsf{out}\})$, and a third by $y : (\{\mathsf{typ}\}, \{\mathsf{out}\})$.

There is one final point to consider before moving on from this example. We began by assuming that, at population time, the controller will *not* install new policies into either x or y that copy the packet. But, suppose the controller *could* install population-time policies that emitted additional packet copies. In this case, the compiler takes three additional steps. First, extra table variables are added to the consolidation stage (p_c), one per table variable in the original policy. Second, table variables in the egress pipeline (p_e) are replaced with fresh table variables. And finally, the compiler produces a transformer θ that applies multicast consolidation to policies destined for x and y at population time, placing the results into the new tables inserted into p_c and p_e.

To capture this formally, we define syntactically restricted forms for the consolidation and egress stages that model consolidated packet duplication and tagging. The consolidation form is similar to the multicast stage presented in Figure 3.7 but slightly higher-level, in that it may contain table variables and additional field modifications—later compilation phases will factor these out.

Definition 5 (Multicast-free CNC)**.** *Let r range over multicast-free CNC policies.*

Definition 6 (Multicast consolidation stages)**.**

$$
\begin{array}{rrcl}
\textit{consolidation sequence} & s & ::= & \Pi_i f_i \leftarrow 1 \mid (x : \tau); \Pi_i f_i \leftarrow 1 \\
\textit{consolidation stage} & m \in \mathsf{M} & ::= & \sum_i a_i; s_i \\
\textit{egress stage} & n \in \mathsf{N} & ::= & \mathsf{id} \mid n; (x : \tau) \mid n; r \\
 & & \mid & n; \mathsf{if}\ \Pi_i f_i = 1\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}
\end{array}
$$

A *consolidation sequence* is a sequence of zero or more modifications to metadata fields, written $\Pi_i f_i \leftarrow 1$, where zero modifications stands for the identity policy id. Each metadata field is linked to an instance of packet duplication (p+q) in the original policy, and the value of the field indicates whether the left or right sub-policy should be applied to the packet—$f \leftarrow 0$ for p, and $f \leftarrow 1$ for q. As metadata fields are initialized to zero, the consolidation sequence only updates metadata fields to 1. The original policy will likely contain table variables, and the controller may install population-time rules into these tables that contain packet duplications. Consolidation sequences can also contain table variables, allowing the transformer to extend the consolidation stage at population time.

The *consolidation stage* is a sum of consolidation sequences, where zero summands indicates the policy drop. Each consolidation sequence corresponds to one copy of the packet, and the modifications in the sequence set the metadata that tags each packet. A predicate a_i guards each consolidation sequence with the conditions in the original policy that lead to this packet duplication. For example, consider just the if statement found in the example policy introduced earlier.

```
if ethtyp = arp then
  if dstmac = 2 then out ← 2 else
  if dstmac = 3 then out ← 3 else
  drop
else
  x : ({dstip}, {out}) + y : ({typ}, {out})
```

In this sub-policy, an ARP packet will not be copied; rather, it will be emitted from either port 1 or port 2, or dropped. A non-ARP packet, on the other hand, will be copied and processed by both x and y. A consolidation stage for this policy will contain three consolidation sequences, one for the true branch, and one for each sub-

policy in the duplication operation in the false branch. But the policy will emit at most two packets, not three, depending on whether the predicate $\mathsf{typ} = arp$ holds on the incoming packet. And so the first two consolidation sequences are guarded by the predicate $\mathsf{typ} = arp$ and the latter two by $\neg(\mathsf{typ} = arp)$, yielding the following consolidation stage.[2]

$$\mathsf{typ} = arp + \neg\mathsf{typ} = arp + \neg\mathsf{typ} = arp; f_2 \leftarrow 1$$

In general, each consolidation sequence corresponds to one sub-policy of a packet duplication operation, and the predicate a_i contains the conditions that must hold for an incoming packet to reach that duplication operation in the original policy.

The *egress stage* is a sequence of duplication-free sub-policies extracted from the original policy. Each clause of the egress stage uses the metadata bits set in the consolidation stage to decide whether a given copy of the packet should be processed by that stage—guarded stages take the form $\mathsf{if}\ \Pi_i f_i = v\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$, and stages with no guard (r) are applied to every packet copy. A packet copy may be processed by more than one egress stage when packet duplication is nested inside packet duplication in the original policy, as the example earlier in this section demonstrates.

The pipeline function (Definition 7, continued in Definition 8) is a syntax-directed policy transformation that consumes a virtual pipeline p and produces a consolidation stage m_{out}, an egress stage n, and a transformer θ. The function also takes an input consolidation stage m_{in}. Intuitively, m_{in} represents the consolidation stage that has been built up "so far" as pipeline executes recursively, and m_{out} includes both m_{in} and any multicast operations in p, and $\mathsf{pipeline}\ s\ p\ m_{in}$ maintains the invariant that $p; m_{in} \equiv m_{out}; n$. As we will see, phrasing the pipeline function to include m_{in} greatly

[2]This is an intermediate consolidation stage generated for this policy fragment; it will be modified to eventually produce the consolidation stage p_c described earlier. Definition 7 describes the algorithm in full.

Definition 7 (Multicast consolidation)**.** *Let* pipeline p *be a function from metadata predictions, policies, and egress stages to policies. The function* pipeline *is defined as follows.*

$$\mathsf{pipeline} :: (\mathsf{Var} \to \mathsf{Nat}) \to \mathsf{Policy} \to \mathsf{M} \to (\mathsf{M} \times \mathsf{N} \times \Theta)$$

$$\begin{array}{lcl}
\mathsf{pipeline}\ s\ p\ \mathsf{drop} & = & (\mathsf{drop}, \mathsf{drop}, \mathsf{id}) \\
\mathsf{pipeline}\ s\ \mathsf{id}\ m & = & (m, \mathsf{id}, \mathsf{id}) \\
\mathsf{pipeline}\ s\ \mathsf{drop}\ m & = & (\mathsf{drop}, \mathsf{drop}, \mathsf{id}) \\
\\
\mathsf{pipeline}\ s\ b\ (\sum_i a_i; q_i) & = & ((\sum_i b; a_i; q_i), b, \mathsf{id}) \\
\\
\mathsf{pipeline}\ s\ f \leftarrow v\ (m + a; q) & = & \mathsf{let}\ (f = v; a') = \mathsf{specialize}\ (f = v)\ a\ \mathsf{in} \\
 & & \mathsf{let}\ m_1, n_1, \theta_1 = \mathsf{pipeline}\ s\ f \leftarrow v\ m\ \mathsf{in} \\
 & & \mathsf{if}\ f = v; a' \equiv \mathsf{drop}\ \mathsf{then} \\
 & & \quad (m_1, n_1, \theta_1) \\
 & & \mathsf{else} \\
 & & \quad ((m_1 + a'; q), f \leftarrow v, \theta_1) \\
\\
\mathsf{pipeline}\ s\ f \leftarrow v\ (m + a; (x : \tau); q) & = & \mathsf{let}\ \tau_1 = \mathsf{typeof}\ (a; (x : \tau); q)\ \mathsf{in} \\
 & & \mathsf{let}\ \theta_1 = (\lambda b, y. \\
 & & \quad \mathsf{let}\ m_1, _, _ = \\
 & & \quad\quad \mathsf{pipeline}\ s\ f \leftarrow v\ (T_b\ a; (x : \tau); q) \\
 & & \quad \mathsf{if}\ y = x_1\ \mathsf{then}\ m_1\ \mathsf{else}\ T_b\ y)\ \mathsf{in} \\
 & & \mathsf{let}\ m_2, n_2, \theta_2 = \mathsf{pipeline}\ s\ f \leftarrow v\ m\ \mathsf{in} \\
 & & ((m_2 + (x_1 : \tau_1)), f \leftarrow v, \theta_2 \circ \theta_1) \\
\\
\mathsf{pipeline}\ s\ x : (\mathsf{R}, \mathsf{W})\ m & = & \mathsf{let}\ fs = s(x)\ \textit{fresh metadata fields}\ \mathsf{in} \\
 & & \mathsf{let}\ \tau = \mathsf{typeof}\ m\ \mathsf{in} \\
 & & \mathsf{let}\ t_m = y : (\mathsf{R}, fs) \cup \tau\ \mathsf{in} \\
 & & \mathsf{let}\ t_n = z : (\mathsf{R} \cup fs, \mathsf{W})\ \mathsf{in} \\
 & & \mathsf{let}\ \theta' = (\lambda b, w. \\
 & & \quad \mathsf{let}\ m', n, \theta = \mathsf{pipeline}\ s\ (T_b\ x)\ (T_b\ m)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ w = y\ \mathsf{then}\ m' \\
 & & \quad \mathsf{else\ if}\ w = z\ \mathsf{then}\ n \\
 & & \quad \mathsf{else}\ T_{\theta\ b}\ w)\ \mathsf{in} \\
 & & (t_m, t_n, \theta')
\end{array}$$

Definition 8 (Multicast consolidation (continued from Definition 7))**.**

$$
\begin{array}{lcl}
\mathsf{pipeline}\ s\ (\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ m & = & \mathsf{let}\ \Sigma_i a_i; q_i, n_1, \theta_1 = \mathsf{pipeline}\ s\ p_1\ m\ \mathsf{in} \\
& & \mathsf{let}\ \Sigma_j a_j; q_j, n_2, \theta_2 = \mathsf{pipeline}\ s\ p_2\ m\ \mathsf{in} \\
& & ((\Sigma_i b; a_i; s_i + \Sigma_j \neg b; a_j; s_j), \\
& & (\mathsf{qualify}\ b\ n_1); (\mathsf{qualify}\ \neg b\ n_2), \theta_1 \circ \theta_2) \\
\mathsf{pipeline}\ s\ (p_1 + p_2)\ m & = & \mathsf{let}\ f = a\ \textit{fresh metadata field}\ \mathsf{in} \\
& & \mathsf{let}\ m_1, n_1, \theta_1 = \mathsf{pipeline}\ s\ p_1\ m\ \mathsf{in} \\
& & \mathsf{let}\ \Sigma_j a_j; q_j, n_2, \theta_2 = \mathsf{pipeline}\ s\ p_2\ m\ \mathsf{in} \\
& & \mathsf{let}\ n_1' = \mathsf{qualify}\ f = 0\ n_1\ \mathsf{in} \\
& & \mathsf{let}\ n_2' = \mathsf{qualify}\ f = 1\ n_2\ \mathsf{in} \\
& & ((m_1 + \Sigma_j a_j; f \leftarrow 1; q_j), n_1'; n_2', \theta_1 \circ \theta_2) \\
\mathsf{pipeline}\ s\ (p_1; p_2)\ m & = & \mathsf{let}\ m_2, n_2, \theta_2 = \mathsf{pipeline}\ s\ p_2\ m\ \mathsf{in} \\
& & \mathsf{let}\ m_1, n_1, \theta_1 = \mathsf{pipeline}\ s\ p_1\ m_2\ \mathsf{in} \\
& & (m_1, n_1; n_2, \theta_2 \circ \theta_1) \\
\mathsf{pipeline}\ s\ (p_1 || p_2)\ m & = & \mathsf{pipeline}\ s\ (p_1; p_2)\ m
\end{array}
$$

Definition 9 (Specialization)**.** *Let* specialize $f = v\ a$ *be the unique homomorphism of Concurrent NetCore defined on primitive programs by:*

$$
\begin{array}{rcl}
g(\mathsf{id}) & \triangleq & \mathsf{id} \\
g(\mathsf{drop}) & \triangleq & \mathsf{drop} \\
g(f = v') & \triangleq & \begin{cases} \mathsf{id} & \textit{if } v = v' \\ \mathsf{drop} & \textit{otherwise} \end{cases} \\
\mathsf{specialize}\ f = v\ p & \triangleq & f = v; g(p)
\end{array}
$$

Definition 10 (Qualify n with a)**.**

$$
\begin{array}{lcl}
\mathsf{qualify} & :: & \mathsf{Predicate} \to \mathsf{N} \to \mathsf{N} \\
\mathsf{qualify}\ a\ \mathsf{id} & = & \mathsf{id} \\
\mathsf{qualify}\ a\ x : \tau & = & \mathsf{if}\ a\ \mathsf{then}\ x : \tau\ \mathsf{else\ id} \\
\mathsf{qualify}\ a\ (n; r) & = & (\mathsf{qualify}\ a\ n); \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else\ id} \\
\mathsf{qualify}\ a\ (n; \mathsf{if}\ b\ \mathsf{then}\ r\ \mathsf{else\ id}) & = & (\mathsf{qualify}\ a\ n); \mathsf{if}\ a; b\ \mathsf{then}\ r\ \mathsf{else\ id}
\end{array}
$$

simplifies the sequencing case ($p = p_1; p_2$). The first call to pipeline takes the empty consolidation stage (id) as input.

Finally, the pipeline function also requires a compile-time estimation s that maps table variables to the maximum number of packet duplications allowed at population time, which is used to reserve space for population-time packet duplication by inserting new table variables matching on reserved metadata fields into consolidation sequences and the egress stage. At population time, the translator will transform policies intended for the original table variable (*i.e.* virtual table) into additional modification sequences for the consolidation sequences and additional extracted policies for the egress stage.

As we explore the details of the pipeline function, our focus will be on producing an m_{out} and n to preserve the equivalence $p; m_{in} \equiv m_{out}; n$, often accompanied by informal reasoning to that effect. We shall prove this equivalence more formally later on, in Theorem 3.

The base cases are straightforward, save for field modification. When $p =$ drop or $m_{in} =$ drop, the policy as a whole is equivalent to drop (by [KA-Seq-Zero] and [KA-Zero-Seq] from Chapter 2), and so the consolidation and egress stages may be drop as well. When $p =$ id, then p commutes with m_{in}. Field matches ($p = f = v$) are prepended to the predicate guarding each consolidation sequence in m_{in}, eagerly dropping packets that do not match. By construction, consolidation stages only modify metadata fields, which do not appear in the original policy, and so field matches commute with m_{in}.

The case where $p = f \leftarrow v$ is made complicated by the presence of the predicates a_i found in consolidation sequences: Our intention is to commute the sequence $f \leftarrow v; m_{in}$, but predicates in m_{in} may depend on the value of the field f being modified. Hence, we must produce a new predicate a_i' that reflects the impact of the field modification on a. This strategy works when a_i exists at compilation time, but

consolidation sequences may also contain table variables to extend the consolidation stage at population time.

The pipeline function handles the non-table case separately. Suppose we have a consolidation sequence $a; q$ (without a table). First, note that $a; q; f \leftarrow v \equiv a; f \leftarrow v; q$, because (by construction) the modifications in consolidation sequences only modify metadata fields, not fields in the policy. If each primitive operation within a predicate commutes with $f \leftarrow v$, then the entire predicate also commutes (Lemma KAT-Commute from Figure 2.5). The specialize function (Definition 9) produces a new predicate a' by replacing each primitive operation in a with id when it matches the same field and value being modified and with drop when it matches the same field but a *different* value. And by again employing the KAT-Commute lemma, it follows that if $a' = \mathsf{specialize}\ f = v\ a$, then $f \leftarrow v; a \equiv a'; f \leftarrow v$. Hence, $a; q; f \leftarrow v \equiv f \leftarrow v; a'; q$. The pipeline function repeats these steps for each consolidation sequence.

On the other hand, suppose that the consolidation contains a table variable: $a; (x : \tau); q$. Rather than specializing a, which cannot be done without knowing the contents of the table variable x, the pipeline function instead *defers* specialization to population time, when the contents of x are known: The consolidation sequence is replaced by a new table variable, x_1, and the transformer θ_1 takes a table binding b (to be supplied at population time) and applies pipeline to the new policy bound in place of the original table variable x, producing a new table binding that installs the resulting consolidation stage when applied to the new table x_1. The same steps are repeated for each consolidation sequence, composing the transformers at each step.

Table variables are another tricky case—we must use the s argument to estimate how much packet duplication may occur at population time and reserve additional metadata fields to support future refactoring. Applied to a table variable, the pipeline function produces a transformer θ that, in turn, compiles all future table updates b—using s to preallocate metadata fields for future updates. A key property of well-

formed table updates is that they produce closed terms—hence, invoking pipeline inside θ on the updated table $b\ x$ runs no risk of divergence.

If statements require special handling. Recall that if a then p else q desugars to $a; p + \neg a; q$. Despite employing the union operator ($+$), if statements never produce multiple packets; rather, at least one branch is guaranteed to drop the packet, as a consequence of the law of excluded middle and the axiom of predicate contradiction, which state that either a predicate or its negation holds (BA-Excl-Mid) but not both (BA-Contra). As there is no packet duplication to consolidate, if statements are treated like filters: the pipeline function is invoked recursively on the branches, the if statement predicate b is prepended to the predicates in the resulting consolidation sequences, and the transformers are composed.

As one might expect, the bulk of the work takes place in the multicast case. Given a policy $p + q$, our strategy is as follows. First, recursively consolidate p and q, producing two consolidation stages that eagerly produce and tag any additional packet copies found within p and q, as well as egress pipelines that process those copies. Then, pick a fresh field f that neither the consolidation nor egress stages of p or q use. For each sequence in the consolidation stage produced from q, set f to 1; metadata fields are initialized to zero, and so no modification of p's consolidation stage is necessary. Finally, add a predicate to each egress pipeline from p with $f = 0$ and from q with $f = 1$—the qualify function (Definition 10) accomplishes this by transforming if a then n else id into an egress pipeline n' with the predicate a conjoined to the guard in each subsequent if statement. The resulting, combined consolidation stage contains all the sequences from p and q, but with an additional tag bit; and the resulting, sequenced egress pipelines differentiate the copies by examining the additional bit. By construction, θ functions extend the domain of table bindings to accommodate new table variables. Hence, we can simply compose the θ functions produced by recursive compilation.

The sequencing case, where $p = p_1; p_2$, simply invokes pipeline recursively first on p_2 and then on p_1. And finally, the concurrency case reduces to the sequential case by Lemma 5; concurrency can be recovered later in the compilation process.

Example. As a further example, let's look at how multicast consolidation works on the $r+m$ fragment of the example policy from the beginning of Chapter 3, reproduced here for convenience.

$$\begin{aligned} r &\triangleq \mathsf{in} = 1; \mathsf{out} \leftarrow 2 + \mathsf{in} = 2; \mathsf{out} \leftarrow 1 \\ m &\triangleq (x : (\{\mathsf{typ}, \mathsf{src}\}, \{\mathsf{out}\})) \end{aligned}$$

Recall that m contains a table variable—and the controller, at population time, may install a policy into m that duplicates some packets. The compiler relies on a hint, s, that pre-allocates metadata fields corresponding to the amount of multicast that future updates may contain. Let $fs = s\ x$ be a set of such fields. The policy produced by pipeline $s\ (r + m)$ id will be

$$\begin{aligned} &(f \leftarrow 0 + f \leftarrow 1; y : (\emptyset, fs)); \\ &(\mathsf{if}\ f = 0\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}); \\ &(\mathsf{if}\ f = 1\ \mathsf{then}\ z : (\{\mathsf{typ}, \mathsf{src}\} \cup fs, \{\mathsf{out}\})\ \mathsf{else}\ \mathsf{id}), \end{aligned}$$

and the binding transformer θ will be

$$\begin{aligned} &(\lambda b, w.\mathsf{let}\ q, r, \theta' = \mathsf{pipeline}\ s\ (T_b\ x)\ \mathsf{in} \\ &\quad \mathsf{if}\ w = y\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{if}\ w = z\ \mathsf{then}\ r\ \mathsf{else}\ T_b\ w). \end{aligned}$$

We introduce a fresh metadata field f to consolidate multicast in a single stage and tag each packet copy, and the remainder of the policy uses the tag to determine whether to apply r or m to each fragment. Because m contains a table variable x,

we also add new tables y and z to handle any multicast that m may contain in the future—and we produce a function θ to ensure this.

Now, suppose a population-time update arrives to x as part of a well-formed table binding, b. Applying $\theta\ b$ to the compiled policy will consolidate any multicast present in b and install appropriate policies in y and z. Since $T_b\ x$ produces a closed policy, θ' (the result of reapplying the $\mathsf{pipeline}$ function within the binding transformer θ) is always the identity function.

Overhead. There are two forms of overhead that may be introduced as part of multicast refactoring: (1) a fresh metadata field is required for each occurrence of packet duplication; and (2) the operands of each packet duplication operation are instrumented to match an extra metadata bit, which may require wider tables at the expense of depth, *i.e.* the number of rules that may be installed. However, the RMT architecture has several predicates on metadata built into the pipeline, such as the "next table" operation. In the future, it would be interesting to explore using these and other mechanisms to reduce the overhead introduced at this stage.

Proof of semantic preservation. Finally, we prove that the original policy is equivalent to the compiled policy for all table updates. We use z to model the fact that metadata is initially assigned a value of 0 when the packet arrives at the switch, and that metadata is not observable once the packet has left the switch.

Theorem 3 (Multicast consolidation preserves semantics)**.** *For all policies p, types $\tau_p = (\mathsf{R}_p, \mathsf{W}_p)$ and $\tau_m = (\mathsf{R}_m, \mathsf{W}_m)$, consolidation stages m_1, metadata annotations s, contexts Γ, and table bindings b, if*

1. *$p \vdash b\ \mathsf{wf}$ and $m_1 \vdash b\ \mathsf{wf}$, and*
2. *$\Gamma \vdash p : \tau_p$ and $\Gamma \vdash m_1 : \tau_m$, and*
3. *$m_2, n, \theta = \mathsf{pipeline}\ s\ p\ m_1$,*

and let $z = \Pi_i f_i \leftarrow 0$*, for all metadata fields* f_i *introduced in* pipeline s p m_1*, then*

1. $m_2 \vdash \theta b$ wf *and* $n \vdash \theta b$ wf*, and*
2. *there exists* Γ' *and* Γ'' *such that*
   - $\Gamma' = \Gamma, \Gamma''$*, and*
   - $\Gamma' \vdash m_2 : (\mathsf{R}_m \cup \mathsf{R}_p \cup \mathsf{W}_p, \mathsf{W}_m \cup \{fs\})$*, and*
   - $\Gamma' \vdash n : (\mathsf{R}_p \cup fs, \mathsf{W}_p)$*, and*
3. $T_b\ z; p; m_1; z \equiv T_{\theta b}\ z; m_2; n; z$.

Proof. The proof proceeds by induction on the structure of p and m_1, relying on Lemmas 4 and 5 and the axioms of NetKAT [2] to establish equivalence. This theorem appears as Theorem 9 in Appendix C and is proved there in full.

□

4.2.2 If De-nesting

The multicast consolidation phase produces an egress pipeline, which is made up of a sequence of if statements. This is convenient, because the RMT pipeline is composed of a series of tables in sequence, making the mapping simpler. Unfortunately, as we saw in the running example, the recursive nature of multicast consolidation may nest egress pipelines generated from sub-policies inside if statement branches. This stage breaks apart if statements with complex policies in each branch into two if statements in sequence, each with a single complicated branch. As such, it allows one large if statement to more easily be split into two tables.

After applying multicast consolidation to the running example p, we have a consolidation stage p_c and an egress pipeline p_e. We will focus on the larger if statement

in p_e.

$$
\begin{array}{l}
\mathsf{if}\ f_1 = 1\ \mathsf{then} \\
\quad \mathsf{if}\ \mathsf{ethtyp} = arp\ \mathsf{then} \\
\quad\quad \mathsf{if}\ \mathsf{dstmac} = 2\ \mathsf{then}\ \mathsf{out} \leftarrow 2\ \mathsf{else} \\
\quad\quad \mathsf{if}\ \mathsf{dstmac} = 3\ \mathsf{then}\ \mathsf{out} \leftarrow 3\ \mathsf{else} \\
\quad\quad \mathsf{else\ drop} \\
\quad \mathsf{else} \\
\quad\quad \left(\begin{array}{l} \mathsf{if}\ f_2 = 0\ \mathsf{then} \\ x : (\{\mathsf{dstip}\}, \{\mathsf{out}\}) \\ \mathsf{else\ id} \end{array}\right) ; \left(\begin{array}{l} \mathsf{if}\ f_2 = 1\ \mathsf{then} \\ y : (\{\mathsf{typ}\}, \{\mathsf{out}\}) \\ \mathsf{else\ id} \end{array}\right) \\
\mathsf{else\ id}
\end{array}
$$

The first step is to combine nested if statements in an entirely standard way.

$$
\begin{array}{l}
\mathsf{if}\ f_1 = 1; \mathsf{ethtyp} = arp\ \mathsf{then} \\
\quad \mathsf{if}\ \mathsf{dstmac} = 2\ \mathsf{then}\ \mathsf{out} \leftarrow 2\ \mathsf{else} \\
\quad \mathsf{if}\ \mathsf{dstmac} = 3\ \mathsf{then}\ \mathsf{out} \leftarrow 3\ \mathsf{else} \\
\quad \mathsf{else\ drop} \\
\mathsf{else\ if}\ f_1 = 1\ \mathsf{then} \\
\quad \left(\begin{array}{l} \mathsf{if}\ f_2 = 0\ \mathsf{then} \\ x : (\{\mathsf{dstip}\}, \{\mathsf{out}\}) \\ \mathsf{else\ id} \end{array}\right) ; \left(\begin{array}{l} \mathsf{if}\ f_2 = 1\ \mathsf{then} \\ y : (\{\mathsf{typ}\}, \{\mathsf{out}\}) \\ \mathsf{else\ id} \end{array}\right) \\
\mathsf{else\ id}
\end{array}
$$

The general algorithm for if statement flattening is as follows.

Definition 11 (If statement flattening)**.**

$$\mathsf{flatten\ if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ \mathsf{else}\ r \quad = \quad \mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else}\ (\mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ r)$$

And, by way of the equational theory, we can show that Definition 11 produces an equivalent policy.

Lemma 7 (If flattening preserves semantics)**.** *For all well-typed predicates a and b and policies p and q and r, the following equivalence holds.*

$$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ \mathsf{else}\ r \equiv \mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else}\ (\mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ r)$$

Proof. The proof goes by a series of applications of the equational axioms. This lemma appears as Lemma 69 in Appendix C, along with the sequence of axioms applied and the transformations they yield.

□

The next step is to split the if statement into two, using a fresh field f_3.

$$\left(\begin{array}{l}\mathsf{if}\ f_1 = 1; \mathsf{ethtyp} = \mathit{arp}\ \mathsf{then}\\ \quad \mathsf{if}\ \mathsf{dstmac} = 2\ \mathsf{then}\ \mathsf{out} \leftarrow 2\ \mathsf{else}\\ \quad \mathsf{if}\ \mathsf{dstmac} = 3\ \mathsf{then}\ \mathsf{out} \leftarrow 3\ \mathsf{else}\\ \quad \mathsf{else}\ \mathsf{drop}\\ \mathsf{else} f_3 \leftarrow 1\end{array}\right); \left(\begin{array}{l}\mathsf{if}\ f_3 = 1; f_1 = 1\ \mathsf{then}\\ \quad \left(\begin{array}{l}\mathsf{if}\ f_2 = 0\ \mathsf{then}\\ x : (\{\mathsf{dstip}\}, \{\mathsf{out}\})\\ \mathsf{else}\ \mathsf{id}\end{array}\right);\\ \quad \left(\begin{array}{l}\mathsf{if}\ f_2 = 1\ \mathsf{then}\\ y : (\{\mathsf{typ}\}, \{\mathsf{out}\})\\ \mathsf{else}\ \mathsf{id}\end{array}\right)\\ \mathsf{else}\ \mathsf{id}\end{array}\right)$$

Definition 12 (If de-nesting)**.** *Let* denest *be a function defined as follows, where f is a fresh field.*

$$\begin{aligned}\mathsf{denest}\ (\mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q) &= (\mathsf{if}\ a\ \mathsf{then}\ p; f \leftarrow 1\ \mathsf{else}\ \mathsf{id}); (\mathsf{if}\ f = 0\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{id})\\ \mathsf{denest}\ \_ &= \bot\end{aligned}$$

At first glance, this transformation makes the policy larger. But note that we have transformed a single term (the original if statement) with two potentially large, complex branches (p and q), into the sequence of two new if statements with p and q distributed between them. Many switching architectures support sequences of tables, but we are not aware of any that support if statements directly; rather, if statements can be compiled to a single table (as per Section 4.1) with some overhead. The sequence of two if statements can be deployed to two tables in sequence with less overhead.

Finally, we can extend if statement flattening to flatten nested sequences of if statements. Applied to our example, it yields the following equivalent policy.

$$\left(\begin{array}{l}\mathsf{if}\ f_1 = 1; \mathsf{ethtyp} = arp\ \mathsf{then}\\ \quad \mathsf{if}\ \mathsf{dstmac} = 2\ \mathsf{then}\ \mathsf{out} \leftarrow 2\ \mathsf{else}\\ \quad \mathsf{if}\ \mathsf{dstmac} = 3\ \mathsf{then}\ \mathsf{out} \leftarrow 3\ \mathsf{else}\\ \quad \mathsf{else}\ \mathsf{drop}\\ \mathsf{else}\ f_3 \leftarrow 1\end{array}\right);$$

$$\left(\begin{array}{l}\mathsf{if}\ f_3 = 1; f_1 = 1; f_2 = 0\ \mathsf{then}\\ \quad x : (\{\mathsf{dstip}\}, \{\mathsf{out}\})\\ \mathsf{else}\ \mathsf{id}\end{array}\right); \left(\begin{array}{l}\mathsf{if}\ f_3 = 1; f_1 = 1; f_2 = 1\ \mathsf{then}\\ \quad y : (\{\mathsf{typ}\}, \{\mathsf{out}\})\\ \mathsf{else}\ \mathsf{id}\end{array}\right)$$

Definition 13 presents the steps necessary to flatten a sequence of if statements. The definition is tailored to the output of the multicast consolidation stage, wherein the false branch of if statements in the egress pipeline is always id.

Definition 13 (If statement sequence flattening)**.**

$$\mathsf{flatten}\left(\begin{array}{l}\mathsf{if}\ a\ \mathsf{then}\\ \quad(\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else\ id});\\ \quad q\\ \mathsf{else\ id}\end{array}\right) = \begin{array}{l}(\mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else\ id});\\ (\mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else\ id})\end{array}$$

$$\mathsf{flatten}\ \_ = \bot$$

Flattening can be applied recursively across the sequence of nested if statements. The correctness of this transformation hinges on the nature of the if statements generated by the compiler; specifically, that the if statement predicates match on metadata introduced in previous compilation stages, which is *not* used in the branches of the if statement.

Lemma 8. *For all well-typed predicates a and b and policies p and q, if $a; p \equiv p; a$, then*

$$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else\ id}); q\ \mathsf{else\ id} \equiv (\mathsf{if}\ a; b\ \mathsf{then}\ q\ \mathsf{else\ id}); \mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else\ id}.$$

Proof. The proof proceeds by a series of transformations guided by the equational theory. This lemma appears as Lemma 71 in Appendix C along with a full proof. □

Together, Definitions 11, 12, and 13 combine to transform egress pipelines produced by multicast consolidation into sequences of flattened, de-nested if statements. Note that the output, although simplified, is still a syntactically valid egress pipeline as defined in Definition 6.

4.2.3 Field Extraction

The RMT architecture also requires that the output port of each packet be set during the multicast stage. Field extraction examines a policy to determine all the conditions under which any modification to a given field may take place, and then rewrites the policy so that modifications to that field happen first. Continuing with our example, suppose we wish to extract modifications to the out field.

$$\left(\begin{array}{l}\textsf{if } f_1 = 0 \textsf{ then}\\ \quad \textsf{out} \leftarrow 1\\ \textsf{else id}\end{array}\right); \left(\begin{array}{l}\textsf{if } f_1 = 1; \textsf{ethtyp} = arp \textsf{ then}\\ \quad \textsf{if dstmac} = 2 \textsf{ then out} \leftarrow 2 \textsf{ else}\\ \quad \textsf{if dstmac} = 3 \textsf{ then out} \leftarrow 3 \textsf{ else}\\ \quad \textsf{else drop}\\ \textsf{else } f_3 \leftarrow 1\end{array}\right);$$

$$\left(\begin{array}{l}\textsf{if } f_3 = 1; f_1 = 1; f_2 = 0 \textsf{ then}\\ \quad x : (\{\textsf{dstip}\}, \{\textsf{out}\})\\ \textsf{else id}\end{array}\right); \left(\begin{array}{l}\textsf{if } f_3 = 1; f_1 = 1; f_2 = 1 \textsf{ then}\\ \quad y : (\{\textsf{typ}\}, \{\textsf{out}\})\\ \textsf{else id}\end{array}\right)$$

There are five points in this example where the out field might be set: once in the first if statement (out $\leftarrow 1$), twice in the second (out $\leftarrow 2$ and out $\leftarrow 3$), once in the third (in x), and once in the fourth (in y). The choice of which of these assignments occurs depends on the predicates guarding them. Hence, field extraction rewrites the policy into two stages: the *extraction* stage tests the incoming packet and applies the appropriate modification, and a second stage comprising an instrumented version of the input policy.

As with multicast consolidation, the field extraction algorithm is also tuned for the sequential case. Let's begin constructing an extraction stage for our running example

by examining the right-most if statement in the sequence.

$$\left(\begin{array}{l} \mathsf{if}\ f_3 = 1; f_1 = 1; f_2 = 1\ \mathsf{then} \\ \quad y : (\{\mathsf{typ}\}, \{\mathsf{out}\}) \\ \mathsf{else}\ \mathsf{id} \end{array}\right)$$

We start by pretending that an "empty" extraction stage exists in the sequence, to the right of the if statement. Our goal is build an extraction stage preceding the if statement that (a) extracts any modifications to the out field, and (b) "pushes" the initial extraction stage from right to left. In this case, that stage is empty, and so we end up with the following policy, with the extraction stage on the left and the instrumented policy on the right.

$$\left(\begin{array}{l} f_3 = 1; f_1 = 1; f_2 = 1; f_4 \leftarrow 1; \\ (y_1 : (\{\mathsf{typ}\}, \{\mathsf{out}\} \cup fs)) \\ + \\ \neg(f_3 = 1; f_1 = 1; f_2 = 1); \mathsf{id} \end{array}\right) ; \left(\begin{array}{l} \mathsf{if}\ f_4 = 1\ \mathsf{then} \\ \quad y_2 : (\{\mathsf{typ}\} \cup fs, \emptyset) \\ \mathsf{else}\ \mathsf{id} \end{array}\right)$$

In the original policy, the table variable y may, at population time, be replaced with a policy that matches on typ and modifies the out field. Hence, we do two things. First, we replace y with a new table, y_2, which can read the same fields as y, along with extra metadata fields fs reserved for use at population time, and can write to the same fields as y, barring the out field—which, in this case, means that y_2 cannot write to any fields at all, although it can still drop packets. Next, we replace the predicate guarding the if statement with a test on a fresh metadata field f_4, the reason for which will be apparent shortly.

The extraction stage contains two clauses connected by union. The first is guarded by the original if statement predicate, testing fields f_3, f_1, and f_2. It also sets the new field f_4 and contains another new table, y_1, which can read the typ field and

write to the out as well as the new metadata fields fs. The metadata field f_4 is not strictly necessary in this case, but it guards against the case where the original if statement predicate depends on the original value of the field being extracted—in that case, moving the field modification before the test would change the semantics of the program.

The second clause of the extraction stage corresponds to the false branch of the original if statement, in which no modification occurred. Because this stage takes place after multicast consolidation, which removes any instances of packet duplication, and because field extraction is semantics preserving, it follows that the clauses of the field extraction stage are disjoint; despite the union operator, only one clause will match an incoming packet.

Field extraction also produces a transformer, θ, which applies field extraction to policies destined for the virtual table y in the original pipeline, placing the results into y_1 and y_2. Field extraction produces the following transformer for this fragment.

$$\theta \triangleq \left(\begin{array}{l} \lambda b, z. \\ \quad \mathsf{let}\ e_1, p, \mathsf{id} = \mathsf{ext}_{\mathsf{out}}\ s\ (T_b\ x)\ \mathsf{id}\ \mathsf{in} \\ \quad \mathsf{if}\ z = y_1\ \mathsf{then}\ e_1 \\ \quad \mathsf{else\ if}\ z = y_2\ \mathsf{then}\ p \\ \quad \mathsf{else}\ T_b(z) \end{array} \right)$$

Here, b is a table binding to be transformed, and z is a table variable to be replaced with the result of the transformed table binding. First, the original virtual table variable, x, is transformed according to the binding b (the result of $T_b\ x$) and supplied to the field extraction algorithm ($\mathsf{ext}_{\mathsf{out}}\ s\ (T_b\ x)\ \mathsf{id}$). If z is y_1, then y_1 is replaced with e_1, the resulting extraction stage. Otherwise, if z is y_2, then y_2 is replaced by p, the instrumented policy. Finally, if z is some other table variable unrelated to this particular transformation, the original binding b is used to transform it.

Applying field extraction to the entirety of the egress pipeline in our running example results in the following extraction stage p_g and instrumented egress pipeline p'_e, taking into account simple optimizations like removing summands that are equivalent to drop from the extraction stage.

$$p_g \triangleq \left(\begin{array}{ll} & f_1 = 0; f_7 \leftarrow 1; \mathsf{out} \leftarrow 1 \\ + & f_1 = 1; \mathsf{ethtyp} = arp; \mathsf{dstmac} = 2; f_3 = 1; f_2 = 0; f_5 \leftarrow 1; f_6 \leftarrow 1; f_8 \leftarrow 1; \\ & \mathsf{out} \leftarrow 2; (x_1 : (\{\mathsf{typ}, \mathsf{dstip}\}, \{\mathsf{out}, f_4\} \cup fs)) \\ + & f_1 = 1; \mathsf{ethtyp} = arp; \mathsf{dstmac} = 3; f_3 = 1; f_2 = 0; f_5 \leftarrow 1; f_6 \leftarrow 1; f_9 \leftarrow 1; \\ & \mathsf{out} \leftarrow 3; (x_1 : (\{\mathsf{typ}, \mathsf{dstip}\}, \{\mathsf{out}, f_4\} \cup fs)) \\ + & \neg(\mathsf{ethtyp} = arp); f_3 = 1; f_1 = 1; f_2 = 0; f_5 \leftarrow 1; \\ & (x_1 : (\{\mathsf{typ}, \mathsf{dstip}\}, \{\mathsf{out}, f_4\} \cup fs)) \\ + & \neg(\mathsf{ethtyp} = arp); \neg(f_3 = 1; f_1 = 1; f_2 = 0); \mathsf{id} \end{array}\right)$$

$$p'_e \triangleq \left(\begin{array}{l} \mathsf{if}\ f_7 = 1\ \mathsf{then} \\ \quad \mathsf{id} \\ \mathsf{else\ id} \end{array}\right); \left(\begin{array}{l} \mathsf{if}\ f_6 = 1\ \mathsf{then} \\ \quad \mathsf{if}\ f_8 = 1\ \mathsf{then\ id\ else} \\ \quad \mathsf{if}\ f_9 = 1\ \mathsf{then\ id\ else} \\ \quad \mathsf{else\ drop} \\ \mathsf{else}\ f_3 \leftarrow 1 \end{array}\right); \left(\begin{array}{l} \mathsf{if}\ f_5 = 1\ \mathsf{then} \\ \quad x_2 : (\{\mathsf{dstip}\} \cup fs, \emptyset) \\ \mathsf{else\ id} \end{array}\right); \left(\begin{array}{l} \mathsf{if}\ f_4 = 1\ \mathsf{then} \\ \quad y_2 : (\{\mathsf{typ}\} \cup fs, \emptyset) \\ \mathsf{else\ id} \end{array}\right)$$

Of course, there is a bit of redundancy in this policy. For example, both branches of the left-most if statement are id, and so the entire if statement could be replaced with id. This, in turn, makes setting the field f_7 unnecessary in the extraction stage. These and other, similar optimizations follow directly from the structure of the equational

theory and can be applied throughout the compilation process. However, we leave a more thorough investigation of such optimizations to future work.

We define a *modification stage* (Definition 14) as a special syntactic form capturing the field modifications extracted from the policy, along with any metadata modifications used as part of the extraction. Specifically, a modification stage is a disjunction of *modification sequences*, where each modification sequence characterizes one potential field modification coupled with associated metadata modifications.

Definition 14 (Modification stage)**.**

$$\begin{array}{llll} \textit{modification sequence} & m & ::= & \Pi_i f_i \leftarrow v_i \\ & & \mid & \Pi_i f_i \leftarrow v_i; (x : \tau) \\ \textit{modification stage} & e \in \mathsf{E} & ::= & \sum_j a_j; m_j \end{array}$$

Each modification sequence corresponds to a trace through the original policy that ends with a modification of the given field. The predicate a_j with each sequence holds the conditions of any if statements along the trace that must be true (or false) for the modification to take place, as well as any predicates that match on the field "upstream" from its modification in the original policy, because the ordering of such matches must be preserved. The modification stage is the sum of all such paths.

As with multicast refactoring, field extraction has two goals: to rewrite a given virtual pipeline to extract any fixed field modifications—*i.e.* modifications built directly into the virtual pipeline—into a fixed modification stage, and to produce a transformer that will similarly extract field modifications installed at population time. The table variable in the modification sequences is inserted as a placeholder that will hold additional modification sequences extracted at population time.

The function $\mathsf{ext}_f\ s\ p\ e$ (Definition 15) performs field extraction. As input, it takes the field to extract (f), a prediction of the number of metadata fields required to support field extraction for each table at population time (s), and the policy (p). The function proceeds by bottom-up traversal of the policy, and takes the modification

stage (e) that has been built up "so far" as the final argument; internally, this allows the function to accumulate modification stages. At the top level, this compilation pass is applied to the policy p with an empty multicast stage ($e = \mathsf{id}$).

The field extraction function produces three artifacts. Given a policy p and a modification stage e, it produces a new policy p' wherein modifications of the field f have been removed, and it produces a new modification stage e' that contains the modifications of f in p. Of course, p may contain table variables, and so the function also produces a transformer θ to extract field modifications from population-time updates—both p' and e' may contain new table variables for θ to populate. Ultimately, the transformed policy should behave the same as the new policy; that is, for all table bindings b that might be produced at population time, it should be the case that $T_b\ p; e \equiv T_{(\theta b)}\ e'; p'$.

The field extraction function is recursive in both the structure of the policy and also the structure of the modification stage. The first few base cases are straightforward: Given a policy or modification stage of drop, as in $\mathsf{ext}_f\ \_\ \mathsf{drop}\ e$ or $\mathsf{ext}_f\ \_\ p\ \mathsf{drop}$, the result is essentially drop with the empty transformer $\theta = \mathsf{id}$. When the policy is id, as in $\mathsf{ext}_f\ \_\ \mathsf{id}\ e$, then the modification stage commutes with the policy.

Suppose we reach a match statement, $p = f' = v$. If this is a predicate on the field being extracted, then the predicate must be preserved before the modification by prepending it to each modification sequence. But matches on other fields are unrelated, and so the match and the modification stage commute. Note that the id policy is always returned when $f = f'$ and $f' = v$ otherwise, as the policy component is independent of the contents of the input modification stage in this case. And as there are no tables in the policy, the transformer is simply the identity function.

The case for table variables is more interesting. Given a table variable x that can read and write sets of fields R and W, we defer further compilation to population time by bundling a second call to ext in a transformer and returning two new table

Definition 15 (Sequenced field extraction)**.** *Let* $\mathsf{ext}_f\ s\ p\ e$ *be a function with the following type.*

$$\mathsf{ext}_f \quad :: \quad (\mathsf{Var} \to \mathsf{Nat}) \to \mathsf{Policy} \to \mathsf{E} \to (\mathsf{E} \times \mathsf{Policy} \times \Theta)$$

The function $\mathsf{ext}_f\ s\ p\ e$ *is defined as follows.*

$$\begin{array}{lcl}
\mathsf{ext}_f\ _\ p\ \mathsf{drop} & = & (\mathsf{drop}, \mathsf{id}, \mathsf{id}) \\
\mathsf{ext}_f\ _\ \mathsf{id}\ e & = & (e, \mathsf{id}, \mathsf{id}) \\
\mathsf{ext}_f\ _\ \mathsf{drop}\ e & = & (\mathsf{drop}, \mathsf{drop}, \mathsf{id}) \\
 & & \\
\mathsf{ext}_f\ s\ b\ (e + a; m) & = & \mathsf{let}\ (\mathsf{R}, _) = \mathsf{typeof}\ b\ \mathsf{in} \\
 & & \quad \mathsf{if}\ f \in \mathsf{R}\ \mathsf{then} \\
 & & \quad\quad \mathsf{let}\ e', _, _ = \mathsf{ext}_f\ s\ b\ e\ \mathsf{in} \\
 & & \quad\quad ((e' + b; a; m), \mathsf{id}, \mathsf{id}) \\
 & & \quad \mathsf{else} \\
 & & \quad\quad ((e + a; m), b, \mathsf{id}) \\
 & & \\
\mathsf{ext}_f\ f' \leftarrow v\ (e + a; \Pi_i f_i \leftarrow v_i) & = & \mathsf{let}\ (f' = v; a') = \mathsf{specialize}\ (f' = v)\ a\ \mathsf{in} \\
 & & \quad \mathsf{let}\ e', p', \theta = \mathsf{ext}_f\ (f' \leftarrow v)\ e\ \ \mathsf{in} \\
 & & \quad \mathsf{if}\ (f' = v; a') \equiv \mathsf{drop}\ \mathsf{then} \\
 & & \quad\quad (e', p', \theta) \\
 & & \quad \mathsf{else\ if}\ f = f'\ \mathsf{then} \\
 & & \quad\quad ((e' + a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i), \mathsf{id}, \theta) \\
 & & \quad \mathsf{else} \\
 & & \quad\quad ((e' + a'; \Pi_i f_i \leftarrow v_i), f' \leftarrow v, \theta) \\
 & & \\
\mathsf{ext}_f\ s\ (f \leftarrow v)\ (e + a; q; (x : \tau)) & = & \mathsf{let}\ e_1 = a; q; (x : \tau)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ \tau_1 = \mathsf{typeof}\ e_1\ \mathsf{in} \\
 & & \quad \mathsf{let}\ p_1 = x_1 : \tau_1 \cup (\emptyset, \{f\})\ \mathsf{in} \\
 & & \quad \mathsf{let}\ \theta_1 = (\lambda b, y. \\
 & & \quad\quad \mathsf{let}\ e_2, p_2, \theta_2 = \\
 & & \quad\quad\quad \mathsf{ext}_f\ s\ (f \leftarrow v)\ (T_b\ e_1) \\
 & & \quad\quad \mathsf{if}\ y = x_1\ \mathsf{then}\ e_2 \\
 & & \quad\quad \mathsf{else}\ T_b\ y)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ e_2, _, \theta_3 = \mathsf{ext}_f\ s\ (f \leftarrow v)\ e\ \mathsf{in} \\
 & & \quad \mathsf{if}\ f = f'\ \mathsf{then} \\
 & & \quad\quad (e_2 + p_1, \mathsf{id}, \theta_3 \circ \theta_1) \\
 & & \quad \mathsf{else} \\
 & & \quad\quad (e_2 + p_1, f' \leftarrow v, \theta_3 \circ \theta_1)
\end{array}$$

Definition 16 (Sequenced field extraction, continued from Definition 15)**.**

$$
\begin{array}{lcl}
\mathsf{ext}_f\ s\ (x : (\mathsf{R}, \mathsf{W}))\ e & = & \mathsf{let}\ fs = s(x)\ \textit{fresh fields}\ \mathsf{in} \\
& & \mathsf{let}\ \tau = \mathsf{typeof}\ e\ \mathsf{in} \\
& & \mathsf{let}\ \theta\ = \\
& & \quad (\lambda b, z. \\
& & \qquad \mathsf{let}\ e_1, p, \mathsf{id} = \mathsf{ext}_f\ s\ (T_b\ x)\ (T_b\ e)\ \mathsf{in} \\
& & \qquad \mathsf{if}\ z = w_1\ \mathsf{then}\ e_1 \\
& & \qquad \mathsf{else\ if}\ z = w_2\ \mathsf{then}\ p \\
& & \qquad \mathsf{else}\ T_b(z))\ \mathsf{in} \\
& & \mathsf{let}\ e' = (w_1 : (\mathsf{R}, f \cup fs) \cup \tau)\ \mathsf{in} \\
& & \mathsf{let}\ p' = (w_2 : (\mathsf{R} \cup fs, \mathsf{W} \setminus f))\ \mathsf{in} \\
& & (e', p', \theta) \\
\\
\mathsf{ext}_f\ s\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ e & = & \mathsf{let}\ f' = a\ \textit{fresh field}\ \mathsf{in} \\
& & \mathsf{let}\ (\sum_i a_{1i}; m_{1i}), p_1, \theta_1 = \mathsf{ext}_f\ s\ p\ e\ \mathsf{in} \\
& & \mathsf{let}\ (\sum_j a_{2j}; m_{2j}), q_2, \theta_2 = \mathsf{ext}_f\ s\ q\ e\ \mathsf{in} \\
& & \mathsf{let}\ e' = \sum_i b; a_{1i}; f' \leftarrow 1; m_{1i} + \\
& & \quad \sum_j \neg b; a_{2j}; m_{2j}\ \mathsf{in} \\
& & (e', \mathsf{if}\ f' = 1\ \mathsf{then}\ p_1\ \mathsf{else}\ q_2, \theta_2 \circ \theta_1) \\
\\
\mathsf{ext}_f\ s\ (p; q)\ e & = & \mathsf{let}\ e_1, q_1, \theta_1 = \mathsf{ext}_f\ s\ q\ e\ \mathsf{in} \\
& & \mathsf{let}\ e_2, p_2, \theta_2 = \mathsf{ext}_f\ s\ p\ e_1\ \mathsf{in} \\
& & (e_2, p_2; q_1, \theta_2 \circ \theta_1) \\
\\
\mathsf{ext}_f\ s\ (p \,||\, q)\ e & = & \mathsf{let}\ e_1, q_1, \theta_1 = \mathsf{ext}_f\ s\ q\ e\ \mathsf{in} \\
& & \mathsf{let}\ e_2, p_2, \theta_2 = \mathsf{ext}_f\ s\ p\ e_1\ \mathsf{in} \\
& & (e_2, p_2; q_1, \theta_2 \circ \theta_1) \\
\\
\mathsf{ext}_f\ \_\ (p + e) & = & \bot
\end{array}
$$

variables (w_1 and w_2), one to hold future modification stages and the other to hold future instrumented policies. Said instrumentation takes place in the transformer θ, which is a function from a table binding b and a table variable z to a policy. Given such a binding at population time, θ invokes field extraction on the policies produced by applying b, producing a new modification stage e_1, an instrumented policy p, and the empty transformer id (which is always produced when compiling closed terms, as produced by well-formed table bindings). The remainder of θ maps the table variables w_1 and w_2 to e_1 and p.

Compiling an if statement, as in $\mathsf{ext}_f\ s\ \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q\ e$, proceeds first by recursing through the branches to produce two new modification stages along with new instrumented policies for the branches. The final policy is produced by replacing the predicate b with a test of a fresh metadata field f', and replacing the branches with the corresponding instrumented versions. And the final modification stage is produced by prepending b to each modification sequence from the true branch, along with a modification $f' \leftarrow 1$ setting the metadata field to 1 (which ensures the proper branch is taken in the instrumented policy). Each sequence in the false branch is prepended with $\neg b$ (metadata fields are initialized to 0 by default), and the final modification stage is the sum of the branch stages.

Sequencing proceeds by straightforward recursion, pushing the results from right to left and composing the resulting transformers. Concurrency proceeds in the same way, but first converting to sequencing. (The concurrency can be recovered later.)

Finally, the case that requires the most detail appears in Definition 16: field modification. Processing a field modification policy also proceeds inductively on the input modification stage, and the two types of modification sequences are handled differently. Sequences that do not contain a table variable can be processed directly. First, recall that our goal is to produce a new policy p' and modification stage e' such

that the equivalence

$$f' \leftarrow v; (e + a; \Pi_i f_i \leftarrow v_i) \equiv e'; p'$$

holds. Reasoning about this equivalence using the equational theory guides the algorithm in this case. For example, note that the modification distributes across the sum, yielding

$$f' \leftarrow v; e + f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i.$$

On the left operand, we invoke $\mathsf{ext}_f\ s\ f' \leftarrow v\ e$ recursively, yielding e', p', θ. And on the right, we make two observations. First, that $f' \leftarrow v \equiv f' \leftarrow v; f' = v$ by the [Mod-Match] axiom. And second, that for any predicate a, we can produce a predicate a' such that $f' = v; a \equiv f' = v; a'$ and $f' = v; a' \equiv a'; f' = v$. The process, called specialize and detailed in Definition 9, produces a' by merging matches on f' in a.

When $f' = v; a'$ is equivalent to the empty policy, the trace corresponding to this modification sequence will drop every packet—perhaps the modification sets one value and the predicate immediately tests the same field for a different value, for example—and this modification sequence can be dropped. Otherwise, if the field modified and the field extracted are the same, then the modification is moved to the modification stage and replaced with id in the policy. Finally, if the fields are different, then the modification remains and the predicate a is replaced with a' in the modification sequence, reflecting the effect of $f' \leftarrow v$.

Modification sequences that contain table variables, such as $e+a; q; (x : \tau)$, require different handling. Recall that in this sequence, a holds the conditions under which modifications appearing in x may occur; q marks the appropriate metadata bits; and x itself will hold population-time updates to the field and to additional metadata. In this step, we build a transformer θ_1 that defers compiling this modification sequence until population time, and we introduce a new table variable x_1 to take its place in the modification stage. If the field modified is the same as the field extracted, then the

modification is replaced with id in the resulting policy; otherwise, the modification remains unchanged.

Overhead. As with multicast refactoring, field extraction introduces overhead in the form of metadata fields and rule space in physical tables: at worst, a fresh metadata field is required for each if statement in the policy. But unlike multicast refactoring, the rule space overhead in field extraction is incurred in the size of the modification stage, which can grow exponentially in the size of the original policy: one modification sequence for each combination of values assigned to the metadata fields.

Proof of semantic preservation. As with multicast consolidation, we show that when metadata has been zeroed at the beginning and end of the policy, the interpretation of the original and compiled forms are equivalent for all table updates.

Theorem 4 (Field extraction preserves semantics)**.** *For all multicast-free policies r, modification stages e, table bindings b, contexts Γ, and fields f, if*

1. $r \vdash b\ \mathsf{wf}$ *and* $e \vdash b\ \mathsf{wf}$*, and*
2. $\Gamma \vdash r : (\mathsf{R}_r, \mathsf{W}_r)$*, and* $\Gamma \vdash e : (\mathsf{R}_e, \mathsf{W}_e)$*, and*
3. $\mathsf{R}_r \cap \mathsf{W}_e = \{f\}$ *and* $\mathsf{W}_r \cap \mathsf{W}_e = \{f\}$*, and*
4. $(e'; r', \theta) = \mathsf{ext}_f\ s\ r\ e$*, and*

and let $z = \Pi_i f_i \leftarrow 0$, for all metadata fields f_i in the set of fresh metadata fields fs introduced in $\mathsf{ext}_f\ s\ r\ e$, then

1. $e' \vdash \theta b\ \mathsf{wf}$ *and* $r' \vdash \theta b\ \mathsf{wf}$*, and*
2. *there exists Γ' and Γ'' such that*
   - $\Gamma' = \Gamma, \Gamma''$*, and*

- $\Gamma' \vdash r' : (\mathsf{R}_r, \mathsf{W}_r \setminus \{f\})$, *and*
- $\Gamma' \vdash e' : (\mathsf{R}_e \cup \mathsf{R}_r, \mathsf{W}_e \cup \{f\} \cup fs)$, *and*

3. $T_b\ (z; r; e; z) \equiv T_{\theta\ b}\ (z; e'; r'; z)$.

Proof. By induction on the structure of r and e, relying on Lemmas 4 and 5 and the axioms of NetKAT [2] to establish equivalence. This theorem appears and is proved in full as Theorem 10 in Appendix C.

□

4.2.4 Table Fitting

At this stage of the compilation process, every occurrence of parallel composition has been consolidated to a single multicast stage, appropriate for deployment to the RMT's multicast stage. What remains are table variables, predicates, field modifications, and if statements joined by sequential and concurrent combinators, as we can see in the egress pipeline of our running example.

$$p'_e \triangleq \begin{pmatrix} \text{if } f_7 = 1 \text{ then} \\ \quad \text{id} \\ \text{else id} \end{pmatrix}; \begin{pmatrix} \text{if } f_6 = 1 \text{ then} \\ \quad \text{if } f_8 = 1 \text{ then id else} \\ \quad \text{if } f_9 = 1 \text{ then id else} \\ \quad \text{else drop} \\ \text{else } f_3 \leftarrow 1 \end{pmatrix}; \begin{pmatrix} \text{if } f_5 = 1 \text{ then} \\ \quad x_2 : (\{\mathsf{dstip}\} \cup fs, \emptyset) \\ \text{else id} \end{pmatrix}; \begin{pmatrix} \text{if } f_4 = 1 \text{ then} \\ \quad y_2 : (\{\mathsf{typ}\} \cup fs, \emptyset) \\ \text{else id} \end{pmatrix}$$

Two tasks remain to match the policy with the architecture model. First, predicates, field modifications, and if statements—which are built into the virtual pipeline and will not change at population time—must nevertheless be replaced by table variables,

as the RMT pipeline has no means of expressing such operations except for populating its tables. A binding transformer will reinstate these policy fragments into the tables at population time. Second, the table variables in the user policy must be fitted to the table variables in the architecture model.

Both steps depend on a second compilation algorithm to compile a table-free user policy to a single physical table; we use the algorithm presented in Section 4.1 for compiling to the physical table format of RMT tables. For the remainder of this section, we refer to this as single-table compilation:

$$\mathsf{single\_table} :: \mathsf{Policy} \to \mathsf{Policy}$$

The resulting policy fits the shape of the matches; crossbar; actions table format described in Figure 3.5.

Table insertion. At configuration time, the RMT switch consists solely of tables arranged via sequential and concurrent composition. Non-table elements in the virtual pipeline are fixed—i.e., they are constant regardless of whichever rules might be installed into the virtual tables at population time. But fixed elements cannot be installed directly on the switch at configuration time. Rather, they must be replaced by table variables, and then reinstalled at population time by a binding transformation. As an example, we can replace our example egress pipeline p'_e with the following policy.

$$p''_e \triangleq \left(\begin{array}{ll} & t_1 : (\{f_7\}, \emptyset) \\ ; & t_2 : (\{f_6, f_8, f_9\}, \{f_3\}) \\ ; & t_3 : (\{f_5, \mathsf{dstip}, fs\}, \emptyset) \\ ; & t_4 : (\{f_4, \mathsf{typ}, fs\}, \emptyset) \end{array}\right)$$

The table insertion algorithm also produces a transformer θ to reinstate the original, fixed policy fragments at population time.

$$
\theta \triangleq \left(\begin{array}{l}
\lambda b, z. \\
\quad \textsf{if } z = t_1 \textsf{ then } \left(\begin{array}{l} \textsf{if } f_7 = 1 \textsf{ then} \\ \quad \textsf{id} \\ \textsf{else id} \end{array}\right) \textsf{ else} \\
\quad \textsf{if } z = t_2 \textsf{ then } \left(\begin{array}{l} \textsf{if } f_6 = 1 \textsf{ then} \\ \quad \textsf{if } f_8 = 1 \textsf{ then id else} \\ \quad \textsf{if } f_9 = 1 \textsf{ then id else} \\ \quad \textsf{else drop} \\ \textsf{else } f_3 \leftarrow 1 \end{array}\right) \textsf{ else} \\
\quad \textsf{if } z = t_3 \textsf{ then } \left(\begin{array}{l} \textsf{if } f_5 = 1 \textsf{ then} \\ \quad (T_b\ x_2) \\ \textsf{else id} \end{array}\right) \textsf{ else} \\
\quad \textsf{if } z = t_4 \textsf{ then } \left(\begin{array}{l} \textsf{if } f_4 = 1 \textsf{ then} \\ \quad (T_b\ y_2) \\ \textsf{else id} \end{array}\right) \textsf{ else} \\
\quad T_b\ z
\end{array}\right)
$$

Definition 17 (Table insertion)**.**

$$
\begin{array}{lcl}
\mathsf{insert\_table} & :: & \mathsf{Policy} \to (\mathsf{Policy} \times \Theta) \\
\\
\mathsf{insert\_table}\ p; q & = & \mathsf{let}\ p', \theta_p = \mathsf{insert\_table}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ q', \theta_q = \mathsf{insert\_table}\ q\ \mathsf{in} \\
 & & (p'; q', \theta_p \circ \theta_q) \\
\mathsf{insert\_table}\ p \mid\mid q & = & \mathsf{let}\ p', \theta_p = \mathsf{insert\_table}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ q', \theta_q = \mathsf{insert\_table}\ q\ \mathsf{in} \\
 & & (p' \mid\mid q', \theta_p \circ \theta_q) \\
\mathsf{insert\_table}\ p & = & (x : \tau, (\lambda b, w. \\
 & & \quad \mathsf{if}\ w = x\ \mathsf{then}\ \mathsf{single\_table}\ (T_b\ p) \\
 & & \quad \mathsf{else}\ T_b\ w))
\end{array}
$$

After completing this step, the transformed user policy consists of table variables and sequential and concurrent combinators. We don't define a case for parallel composition, because any packet duplication is extracted as part of multicast consolidation, and any remaining instances of the union operator must be within if statements—which are treated as any other fixed element and replaced with a table.

Recovering concurrency. During multicast consolidation and field extraction, some instances of concurrent composition were replaced with sequential composition—the egress stage, in particular, comprises a sequence of multicast-free policies, some of which may be able to operate concurrently. The type system indicates which instances of sequential composition can be replaced with concurrent composition: sequentially composed policies with non-overlapping write sets and non-overlapping read/write sets. That is, consider again our table-fied egress

pipeline.

$$p_e'' \triangleq \left(\begin{array}{ll} & t_1 : (\{f_7\}, \emptyset) \\ ; & t_2 : (\{f_6, f_8, f_9\}, \{f_3\}) \\ ; & t_3 : (\{f_5, \mathsf{dstip}, fs\}, \emptyset) \\ ; & t_4 : (\{f_4, \mathsf{typ}, fs\}, \emptyset) \end{array}\right)$$

Because none of these tables write to the same fields, nor do any write to a field another might read, they can all be arranged concurrently rather than sequentially, as per Lemma 5.

$$p_e''' \triangleq \left(\begin{array}{ll} & t_1 : (\{f_7\}, \emptyset) \\ \emptyset||_{\{f_3\}} & t_2 : (\{f_6, f_8, f_9\}, \{f_3\}) \\ {\scriptstyle \{f_3\}}||_{\emptyset} & t_3 : (\{f_5, \mathsf{dstip}, fs\}, \emptyset) \\ \emptyset||_{\emptyset} & t_4 : (\{f_4, \mathsf{typ}, fs\}, \emptyset) \end{array}\right)$$

The compiler can recover concurrency by structurally traversing the compiled policy, searching for and replacing amenable sequential combinators.

Table fitting. Single-table compilation comes with a cost—the number of rules in the compiled table can grow exponentially with the number of sequential combinators in the original policy. However, thanks to the concurrency inherent within physical tables, the policy $p \,||\, q$ does not incur any overhead when installed in a single table. This leads to a choice. Suppose we have a policy $(p; (q \,||\, r)) : \tau$ that we would like to compile to a sequence of two tables, $(x_1 : \tau); (x_2 : \tau)$. Recall that concurrency is commutative (Lemma 4) and equivalent to sequential composition (Lemma 5).

Hence, there are four ways we might compile this policy.

Compilation grouping	Cost
$(p); (q \mid\mid r)$	$(\|p\|) + (\|q\| + \|r\|)$
$(p; q); (r)$	$(\|p\| * \|q\|) + (\|r\|)$
$(p; r); (q)$	$(\|p\| * \|r\|) + (\|q\|)$
$(p; q; r)$	$(\|p\| * \|r\| * \|q\|)$

In the first case, p is compiled to x_1 and $q \mid\mid r$ to x_2. The cost of p (written $|p|$) refers to the number of TCAM or SRAM rows the compiled policy fills. The cost of placing q and r in the same table is $|q| + |r|$. In the next, the division is $p; q$ and r, and here, the cost of placing p and q in the same table is multiplicative in their sizes. Similarly, $p; r$ might be placed in x_1 and q in x_2 at the cost $|p| * |r| + |q|$. Finally, the RMT chip has the capability to join its physical stages together to emulate a single, larger "logical stage." That capability provides a final option, which is to compile p, q, and r to a single table (paying the largest overhead of $|p| * |q| * |r|$). If p, q, and r are of equal size, then the first option is most efficient. But when p is small and x_1 has space remaining, it may make sense to pay the cost of compiling $p; q$ or $p; r$ to x_1. The RMT "logical table" feature is suitable for cases in which p, q, or r are too large to fit in a single physical table. The RMT chip has a limited number of bits a table can match and the number of rules it can hold—each match stage has sixteen blocks of 40b by 2048 entry TCAM memory and eight 80b by 1024 entry SRAM blocks—so deciding how to partition a policy into tables matters.

Since there are many choices about how to fit a collection of tables, we have defined a dynamic programming algorithm to search for the best one. The goal of the algorithm is to fit a well-typed policy, without parallel composition, into as few tables as possible.

Definition 18 (TCAM cost measurement)**.**

$$
\begin{array}{lcl}
\mathsf{table\_cost} & \in & \mathsf{Var} \to \mathbb{N} \\
\mathsf{height}\ x & = & \mathsf{table\_cost}\ x \\
\mathsf{height}\ p; q & = & \mathsf{height}\ p * \mathsf{height}\ q \\
\mathsf{height}\ p \,||\, q & = & \mathsf{height}\ p + \mathsf{height}\ q \\
\mathsf{blocks}\ p & = & \lceil (\mathsf{width}\ p)/40 \rceil * \lceil (\mathsf{height}\ p)/2048 \rceil
\end{array}
$$

As input, the algorithm relies on a user-supplied annotation estimating the maximum size of each user table at population time, written table_cost x. We also rely on several utility functions. The width of a policy (width p) returns the number of bits it matches, while the height (height p) uses the user-supplied annotation to gauge the number of entries that will ever be installed into the policy at population time. Together, they calculate the number of TCAM blocks necessary to implement a policy (blocks p). Similar measurements exist for compiling to SRAM, but we focus on TCAM here.

As input, the algorithm also takes a policy containing only sequences of tables. The policy AST is a tree with combinators at the nodes and tables at the leaves. We need to flatten this tree into the RMT pipeline. To do so, we must consider different groupings of the tree's fringe. For convenience, let t_{ij} represent an in-order numbering of the leaves of the abstract syntax tree, starting with t_{11} as the leftmost leaf. For example, given a policy $(x : \tau_x); (y : \tau_y); (z : \tau_z)$, then t_{23} would be $(y : \tau_y); (z : \tau_z)$.

The algorithm proceeds by building a table m, where each cell $m[i, j]$ holds the smallest number of tables into which the sequence t_{ij} can fit. The crux of the algorithm lies on line 10. Given a sequence t_{ij} for which the optimal fit for each subsequence has been computed, either the entire sequence may be compiled to a single logical table that can be deployed across $\lceil \mathsf{blocks}\ t_{ij}/16 \rceil$ physical tables, or there exists a partitioning t_{ik}, t_{kj} where both subsequences fit into sets of tables, and so the entire

```
   input  : A sequence of t_{1n}
   input  : table_cost
 1 let m[1...n, 1...n] and s[1...n-1, 2...n] be new tables;
 2 for i = 1 to n do
 3 |  m[i,i] = ⌈(blocks t_i)/16⌉;
 4 end
 5 for l = 2 to n do
 6 |  for i = 1 to n - l + 1 do
 7 |  |  j = i + l - 1;
 8 |  |  m[i,j] = ∞;
 9 |  |  for k = i to j - 1 do
10 |  |  |  q = min(m[i,k] + m[k+1,j], ⌈blocks t_{ij}/16⌉);
11 |  |  |  if q < m[i,j] then
12 |  |  |  |  m[i,j] = q;
13 |  |  |  |  s[i,j] = k;
14 |  |  |  end
15 |  |  end
16 |  end
17 end
18 return m and s
```

Algorithm 1: Table fitting.

sequence fits into the sum of the size of the sets. The algorithm contains three nested loops iterating over t_{1n}, giving it a complexity on the order of $\mathcal{O}(n^3)$, where n is size of the policy AST's fringe. The table s records the best partition chosen at each step, from which we can reconstruct the sets of subsequences to compile to each table.

It remains to convert a user policy with concurrent and sequential composition to one without concurrent composition. We apply a brute-force approach. For each concurrent operator $p \,||\, q$, produce two sequences, $p; q$ and $q; p$. Apply Algorithm 1 to each, and select the smallest result. There are on the order of $\mathcal{O}(2^m)$ sequences, where m is the number of concurrency operators, and so this final determinization step runs in $\mathcal{O}(2^m n^3)$. Fortunately, in our experience, policies tend to have on the order of tens of tables, although the tables themselves may hold many more rules.

4.2.5 Combining the Compilation Passes

To recap, we have four compilation stages. Multicast consolidation transforms an initial virtual pipeline into a consolidation phase (with multicast) sequenced with an egress pipeline (without multicast). If de-nesting simplifies nested if statements. Field extraction takes a virtual pipeline without multicast and a field f to extract and returns a modification stage (with modifications to the field f) sequenced with an instrumented policy (without modifications to f). Finally, table fitting takes a virtual policy without multicast, recovers any concurrency inherent in the pipeline, replaces fixed pipeline fragments with tables, and fits the tables to the RMT pipeline structure.

The stages are composed in this order: multicast consolidation, field extraction, and then table fitting. Field extraction is applied to the egress pipeline built during multicast consolidation, and if de-nesting is an optimization that can be applied after each phase.

$$\begin{aligned} m, n, \theta_1 &= \mathsf{pipeline}\ s_1\ p\ \mathsf{id} \\ e, n', \theta_2 &= \mathsf{ext}_{\mathsf{out}}\ s_2\ n\ \mathsf{id} \end{aligned}$$

At this point, we began with a policy p and now have an equivalent policy (modulo metadata) with a consolidation stage followed by a modification stage and an instrumented egress pipeline.

$$p \equiv m; e; n'$$

And, as field extraction was applied to the egress pipeline produced by multicast consolidation, the transformers similarly compose ($\theta_1 \circ \theta_2$). Next, table fitting transforms the egress pipeline n' into the physical table format.

$$p \equiv m; e; \Pi_i \mathsf{physical}_i$$

It remains to put the consolidation and modification stages $(m; e)$ into a form that can be deployed to the multicast stage of the RMT pipeline.

Composing multicast consolidation with field extraction (on the out field) produces two large summations m and e. Together, m and e represent all the ways an incoming packet may be copied and assigned an output port. Our goal is to refactor $m; e$ to fit into the multicast stage in the RMT pipeline. Recall from Section 3.2 that the multicast stage has the following form.

$$\mathsf{multicast} = \sum_i out_i = i; f_{\mathsf{tag}} \leftarrow v_i; \mathsf{out} \leftarrow i$$

Unfortunately, the multicast stage only matches on a specific set of metadata fields (out_i). The consolidation and modification stages may match any field, and so they cannot fit into the multicast stage alone. Instead, we can transform $m; e$ into three new stages: one that sets the appropriate out_i metadata to initiate packet duplication (the *setup* stage), a second that checks the metadata, performs duplication, and sets a unique tag on packet copies (the *multicast* stage), and a third that checks the packet tag, sets the metadata introduced by the compiler, and applies tests from m and e to filter any extra copies (the *check* stage).

The algorithm for combining m and e aligns especially closely with its proof of correctness—both are guided by an application of the equational theory from Chapter 2—and so we present them here together. First, let z as the policy fragment that zeroes metadata. (The collections of metadata fields *out* and t will be explained shortly.)

$$z = out \leftarrow 0; \mathsf{out} \leftarrow 0; t \leftarrow 0$$

We start with the consolidation and extraction stages sequenced, with metadata set to zero before and after execution.

$$z; m; e; z$$

For now, suppose that neither stage contains a table variable. m and e expand as follows.

$$\equiv z; (\sum_i a_i; s_i); (\sum_j b_j; m_j); z$$

By construction, each extraction stage contains a modification to the out field: $m_j \equiv m'_j; \mathsf{out} \leftarrow v_j$. Substituting equals for equals, we have the following.

$$\equiv z; (\sum_i a_i; s_i); (\sum_j b_j; m'_j; \mathsf{out} \leftarrow v_j); z$$

By the distributivity axiom Dist-L, summands with similar output port modifications can be grouped.

$$\equiv z; (\sum_i a_i; s_i); (\sum_k (\sum_j b_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

Also following the distributivity axioms Dist-L and Dist-R, the summation constituting m distributes to the right.

$$\begin{aligned} &\equiv \quad z; (\textstyle\sum_k (\sum_i a_i; s_i); (\sum_j b_j; m'_j); \mathsf{out} \leftarrow v_k); z \\ &\equiv \quad z; (\textstyle\sum_k (\sum_{ij} a_i; s_i; b_j; m'_j); \mathsf{out} \leftarrow v_k); z \end{aligned}$$

By construction, neither m nor e use the collection of metadata fields *out*, and z contains $out \leftarrow 0$. We will now make use of *out*, which is an n-bit metadata field where n is the number of output ports. We write out_i for the i-th bit of *out*. Following the PA-Mod-Mod axiom, we can preface z with arbitrary modifications $\Pi_k out_k \leftarrow 1$,

which then distribute left.

$$\equiv z; (\sum_k (\Pi_k out_k \leftarrow 1); (\sum_{ij} a_i; s_i; b_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

Using PA-Mod-Filter and PA-Mod-Filter-Comm we can create a new test on out_k in each k summand.

$$\equiv z; (\sum_k (\Pi_k out_k \leftarrow 1); out_k = 1; (\sum_{ij} a_i; s_i; b_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

And, by Dist-L, the sequence of modifications to out_k can be extracted to the left.

$$\equiv z; (\Pi_k out_k \leftarrow 1); (\sum_k out_k = 1; (\sum_{ij} a_i; s_i; b_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

At this point, we have extracted the setup stage $p_1 = (\Pi_k out_k \leftarrow 1)$. The next step is to form the multicast and check stages. Recall that the inner summation over ij is grouped by common modifications to the out port; intuitively, each summand represents a different copy that may be sent out the same port k. The metadata field t will allow us to differentiate copies destined to the same port. t is n bits wide, where n is the maximum number of packets that may be sent to the same output port.

Again by PA-Mod-Mod, we can create arbitrary modifications to t from z. And, as t is fresh with respect to the rest of the policy, its modifications commute and distribute left. We can use PA-Mod-Mod again to inject a unique modification to t into each grouped summand.

$$\equiv z; (\Pi_k out_k \leftarrow 1); (\sum_k out_k = 1; (\sum_{ij} t \leftarrow v_{ijk}; a_i; s_i; b_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

Next, note that out itself is a metadata field, and as such is initialized to zero. We can use the $\mathsf{specialize}$ $f$unction of Definition 9 (along with Lemma 65) to resolve matches

on out in each a_i and b_j. Let a'_i and b'_j represent the specialized versions of a_i and b_j.

$$\equiv z; (\Pi_k out_k \leftarrow 1); (\sum_k out_k = 1; (\sum_{ij} t \leftarrow v_{ijk}; a'_i; s_i; b'_j; m'_j); \mathsf{out} \leftarrow v_k); z$$

Conveniently, specialization removes references to the field being specialized. Hence, by Lemma 63, a'_i and b'_j commute with s_i, m'_j, and $\mathsf{out} \leftarrow v_k$, and so we can distribute the out modification (eliminating the grouping) and move $a'_i; s_i; b'_j; m'_j$ to the end of each summand.

$$\equiv z; (\Pi_k out_k \leftarrow 1); (\sum_{ij} out_k = 1; t \leftarrow v_{ijk}; \mathsf{out} \leftarrow v_k; a'_i; s_i; b'_j; m'_j); z$$

The combination of t and out constitutes a unique tag for each packet duplicated in the summation. It turns out that a single summation can be split into two summations, provided that each summand in the original contains a unique tag (Lemma 74). Hence, by PA-Mod-Filter and Lemma 74, we have the following.

$$\begin{aligned} \equiv \quad & z; (\Pi_k out_k \leftarrow 1); \\ & (\textstyle\sum_{ij} out_k = 1; t \leftarrow v_{ijk}; \mathsf{out} \leftarrow v_k); \\ & (\textstyle\sum_{ij} t = v_{ijk}; \mathsf{out} = v_k; a'_i; s_i; b'_j; m'_j); z \end{aligned}$$

The second summation exactly matches the form of the RMT $\mathsf{multicast}$ stage.

The final summation quite nearly resembles an if statement; intuitively, each summand corresponds to a set of packets that will be handled by the same path through the original policy; as such, the tests across summands do not intersect. By lifting specializing of the field extraction tests in b'_j to the modifications in s_1 (Lemma 65),

we can group all the predicates together in each summand.

$$\begin{aligned}
\equiv \quad & z; (\Pi_k out_k \leftarrow 1); \\
& (\textstyle\sum_{ij} out_k = 1; t \leftarrow v_{ijk}; \mathsf{out} \leftarrow v_k); \\
& (\textstyle\sum_{ij} t = v_{ijk}; \mathsf{out} = v_k; a'_i; b''_j; s_i; m'_j); z
\end{aligned}$$

And, following Lemma 75, the final summation can be replaced with an equivalent if statement, forming the check stage.

$$\begin{aligned}
\equiv \quad & z; (\Pi_k out_k \leftarrow 1); \\
& (\textstyle\sum_{ij} out_k = 1; t \leftarrow v_{ijk}; \mathsf{out} \leftarrow v_k); \\
& \mathsf{if}\ t = 1; \mathsf{out} = 1; a'_1; b''_1\ \mathsf{then}\ s_1; m'_1\ \mathsf{else} \\
& \mathsf{if}\ t = 2; \mathsf{out} = 1; a'_1; b''_2\ \mathsf{then}\ s_1; m'_2\ \mathsf{else} \\
& \ldots
\end{aligned}$$

Of course, summands within the consolidation and extraction stages may contain table variables. In that case, three new table variables are created, one each for the setup, multicast, and check stages, and a θ transformer fills those tables at population time by applying the transformation described above.

After combining the consolidation and extraction stages, we have the following (omitting the metadata zeroing), where p_1 is the setup stage and p_3 the check stage.

$$m; e \equiv p_1; \mathsf{multicast}; p_2$$

We can apply table fitting to p_1 and $p_2; n'$ independently to fit them to the RMT pipeline. Altogether, we have the following.

$$
\begin{array}{lcll}
p & \equiv & m; n & (\textit{multicast consolidation}) \\
 & \equiv & m; e; n' & (\textit{field extraction}) \\
 & \equiv & p_1; \mathsf{multicast}; p_2; n' & \text{(merging consolidation and extraction stages)} \\
 & \equiv & \mathsf{physical}_1; \mathsf{multicast}; \Pi_i \mathsf{physical}_i & \text{(table fitting)}
\end{array}
$$

Optimality. The table fitting algorithm is optimal in the sense that it partitions a virtual pipeline—restricted to virtual tables, composed sequentially or concurrently—into the fewest number of physical tables. But compilation as a whole is not optimal. The stages produced by multicast consolidation and field extraction may have equivalent, syntactically smaller forms, as might the original virtual pipeline.

The equational theory has many axioms that equate policies of different syntactic sizes, such as [KA-Dist-L], which relates a term with five sub-terms (p, q, r, sequencing, and union) with an equivalent term with seven sub-terms (in which p and the sequencing operator appear twice). In principle, it may be possible to algorithmically determine the smallest equivalent policy; this would be an interesting avenue for future work.

But even so, given a family of equivalent virtual pipelines, it's not clear that the virtual pipeline with the fewest terms will necessarily compile to the fewest tables. Nor is it clear what optimizations, if any, might result in smaller consolidation, modification, and egress stages at compilation time. Chapter 7 discusses different kinds of optimizations as future work.

Chapter 5

Implementation and Evaluation

We have implemented a prototype of the Concurrent NetCore language as an embedded combinator library in OCaml, along with functions implementing the operational semantics and pipeline compilation algorithms. Evaluating pipeline compilation on two control applications—a learning switch and a stateful firewall—shows that while overhead is non-negligible, pipeline compilation yields improvements over single-table compilation.

5.1 Implementation

The policy language is represented by an abstract syntax tree using OCaml algebraic data types. Packet trees are similarly embedded, and the operational semantics is faithfully implemented as a function that steps a pair of a policy and packet tree. Each of the syntactically-restricted stages associated with the compilation phases takes the form of a unique data type, which each compilation phase builds while traversing the policy abstract syntax tree.

Although this dissertation proves pipeline compilation correct, we nevertheless implemented a regression test suite to aid development of the prototype, consisting of sixty-five tests ranging from unit to system-wide coverage. Many of the tests take

the form of translation validation, where a policy is compiled and both the source and compiled versions are evaluated against a set of input packets, checking for output equivalence.

5.2 Evaluation Setup

To evaluate the CNC compilation algorithms, we implemented two simple applications in CNC—a learning switch and a stateful firewall—and compared the overhead of pipeline compilation to that of single-table compilation. In this setting, overhead is measured as the syntactic size of the compiled policy compared to that of the original.

Learning switch. A learning switch learns the locations of new hosts as they join the network. The controller accomplishes this by monitoring traffic sent from new hosts; if a packet from an unrecognized MAC address arrives at a learning switch, it sends a copy of the packet to the controller, which then updates the forwarding rules on the switch to send traffic destined to that MAC address out the port on which the traffic arrived. If the original packet is destined for a known address—*i.e.* one that has previously sent traffic through the switch—it is forwarded to the appropriate port; otherwise, it is dropped.

Intuitively, the switch performs two tasks simultaneously: monitoring traffic from unknown hosts and forwarding traffic to known hosts. Hence, we can build a virtual pipeline with two tables, one for monitoring, and another for forwarding.

$$learning \triangleq (mon : (\{\mathsf{srcmac}\}, \{\mathsf{out}\})) + (fwd : (\{\mathsf{dstmac}\}, \{\mathsf{out}\}))$$

Here, we model the act of sending the packet to the controller as forwarding the packet out a special port, and so both tables potentially modify the out field. Notably, the union operator allows for a more modular implementation: the monitoring and

forwarding policies can be developed independently, because each operates on its own logical copy of the packet.

The monitoring policy initially sends all packets to the controller (on the special port c), and the forwarding policy begins by dropping all traffic.

$$\begin{aligned} mon_{init} &\triangleq \mathsf{out} \leftarrow \mathsf{c} \\ fwd_{init} &\triangleq \mathsf{drop} \end{aligned}$$

Suppose a new host joins the network with MAC address 1. After the switch sees its first packet, say on port 1, the controller updates the monitoring and forwarding tables.

$$\begin{aligned} mon_1 &\triangleq \mathsf{if}\ \mathsf{srcmac} = 1\ \mathsf{then}\ \mathsf{drop}\ \mathsf{else}\ mon_{init} \\ fwd_1 &\triangleq \mathsf{if}\ \mathsf{dstmac} = 1\ \mathsf{then}\ \mathsf{out} \leftarrow 1\ \mathsf{else}\ fwd_{init} \end{aligned}$$

Both tables continue to grow as hosts join the network.

Stateful firewall. A stateful firewall separates trusted and untrusted hosts. Initially, the firewall blocks all traffic from untrusted hosts; but, after a trusted host initiates a connection to an untrusted host, the firewall adds the remote host to a list of safe external hosts, which are allowed to send traffic in the reverse direction.

As with the learning switch, a stateful firewall comprises two parts—a monitoring element and a forwarding element—which we can implement using a virtual pipeline with two tables joined by union. In this example, we assume that trusted hosts are connected via port 1, and untrusted hosts via port 2.

$$\begin{aligned} &fireWall \triangleq \\ &\mathsf{in} = 1; (mon : (\{\mathsf{dstip}\}, \{\mathsf{out}\})) + \left(\begin{array}{l} \mathsf{if}\ \mathsf{in} = 1\ \mathsf{then} \\ \quad \mathsf{out} \leftarrow 2 \\ \mathsf{else} \\ \quad (fwd : (\{\mathsf{srcip}\}, \emptyset)); \mathsf{out} \leftarrow 1 \end{array} \right) \end{aligned}$$

On the left, all traffic arriving from a trusted host may be monitored to detect new flows to untrusted hosts, while on the right, all traffic from trusted hosts can be forwarded to untrusted hosts, but traffic from untrusted hosts is first checked against the set of allowed connections. The monitoring table initially sends all packets to the controller, while the forwarding table drops all untrusted traffic.

$$\begin{aligned} mon_{init} &\triangleq \mathsf{out} \leftarrow \mathsf{c} \\ fwd_{init} &\triangleq \mathsf{drop} \end{aligned}$$

After a trusted host initiates a new connection, say from a source IP address of 1 to an untrusted host with address 2, the tables are updated as follows.

$$\begin{aligned} mon_0 &\triangleq \mathsf{if\ dstip} = 2\ \mathsf{then\ drop\ else}\ mon_{init} \\ fwd_0 &\triangleq \mathsf{if\ srcip} = 2\ \mathsf{then\ id\ else}\ fwd_{init} \end{aligned}$$

Both tables continue to grow as trusted hosts initiate new connections.

Evaluation setup. Each benchmark is made up of a virtual pipeline and a table binding that holds a population-time update. In order to compare the pipeline compilation algorithms with single table compilation, we take two steps.

- We begin by installing the update to the virtual pipeline and measuring how large the resulting policy would be if the entire populated virtual pipeline were compiled to a single table. The RMT pipeline supports deploying a logical table across many physical tables, and so this represents a straw-man solution: compile complicated pipelines to a single table, and then spread that table across the RMT pipeline.

- Next, we compile the virtual pipeline using the algorithms from Chapter 4, using the size of the population-time update as the "size estimate" required by the compiler. The output is a pipeline that matches the RMT model from

Section 3.2, along with a transformer. We then install the population-time update into the compiled pipeline using the transformer and measure its size.

We scale the size of the population-time update with the number of network events—new hosts, in the case of the learning switch, and new outbound connections for the stateful firewall—and compare the sizes of the resulting policies syntactically, counting the number of policy operators, which accounts for the number of rules as well as the size of each rule.

The experiments were run on a Macbook Pro with a 2.6 GHz Intel Core i7 processor, 16 GB 1600 MHz DDR3 RAM, running OS X 10.9.5. Each benchmark was compiled to a native binary using OCaml version 4.02.1.

5.3 Evaluation Results

The results of our experiments are given in Figure 5.1, which compares the overhead of pipeline compilation to that of single-table compilation, where pipeline compilation consists of multicast consolidation, if statement de-nesting, field extraction of the `out` field, and table fitting, targeting the RMT pipeline model.

Notably, pipeline compilation scales better than single-table compilation, despite the additional machinery that makes pipeline compilation possible—this shows that pipeline compilation is more efficient than compiling a policy into a single table and spreading that table across the physical pipeline. But the evaluation also demonstrates an unfavorable interaction between pipeline compilation phases, where multicast consolidation introduces unnecessary work for field extraction. It also lays bare a serious limitation in the scalability of the prototype implementation, due to problems with memory allocation. The remainder of this section examines these points in more detail.

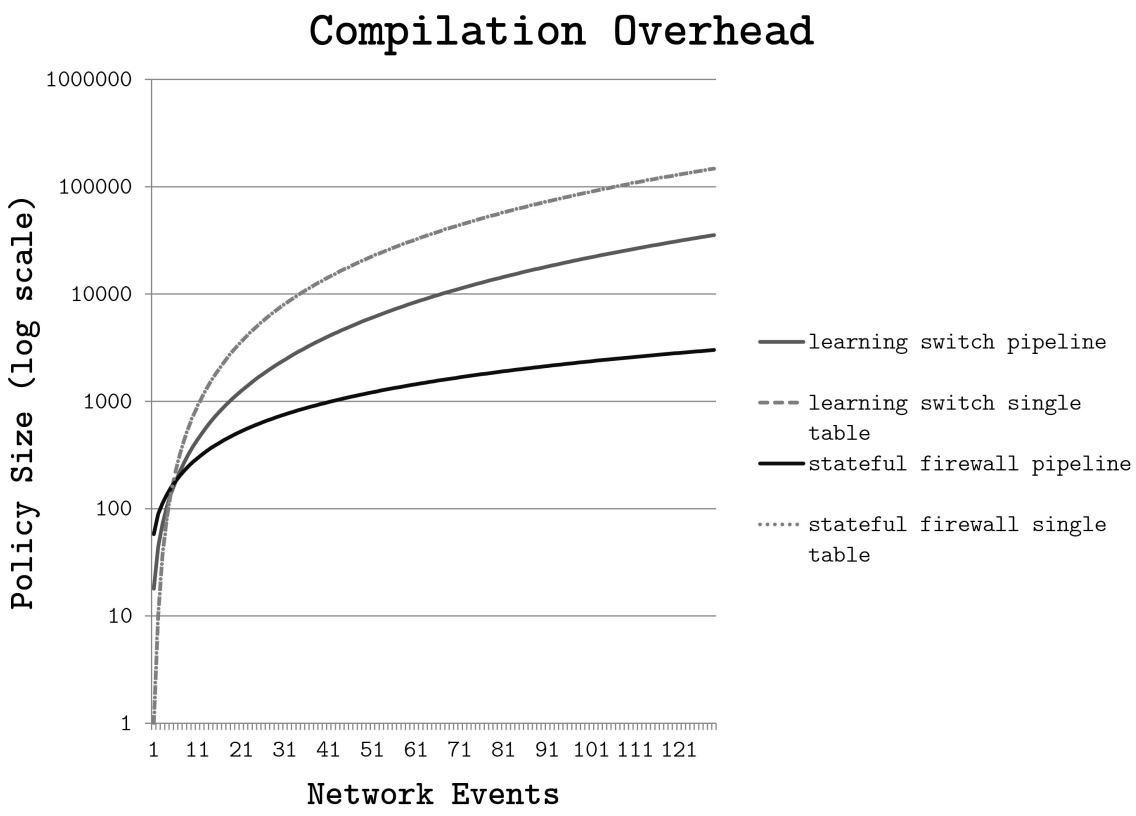

Figure 5.1: The size of compiled virtual pipelines filled with a compiled population-time update. Size measures the number of operators in the policy.

Interaction between compilation phases. Figure 5.2 shows a more detailed breakdown of the learning switch benchmark. The X-axis indicates the number of new hosts that have sent traffic through the learning switch, and the Y-axis is the syntactic size of the policy in logarithmic scale. The lightest line (labeled "input"), closest to the bottom, shows the size of the virtual policy and the population-time update generated by the network events, as described in Section 5.2—it grows linearly with the number of network events. The line above it corresponds to the size of the policy emitted after multicast consolidation (labeled "MC" in the legend). Because the virtual pipeline only contains one instance of multicast, the overhead is minimal.

But the multicast consolidation stage rearranges the policy into a pipeline format. The field extraction stage, which moves modifications to the `out` field earlier in the

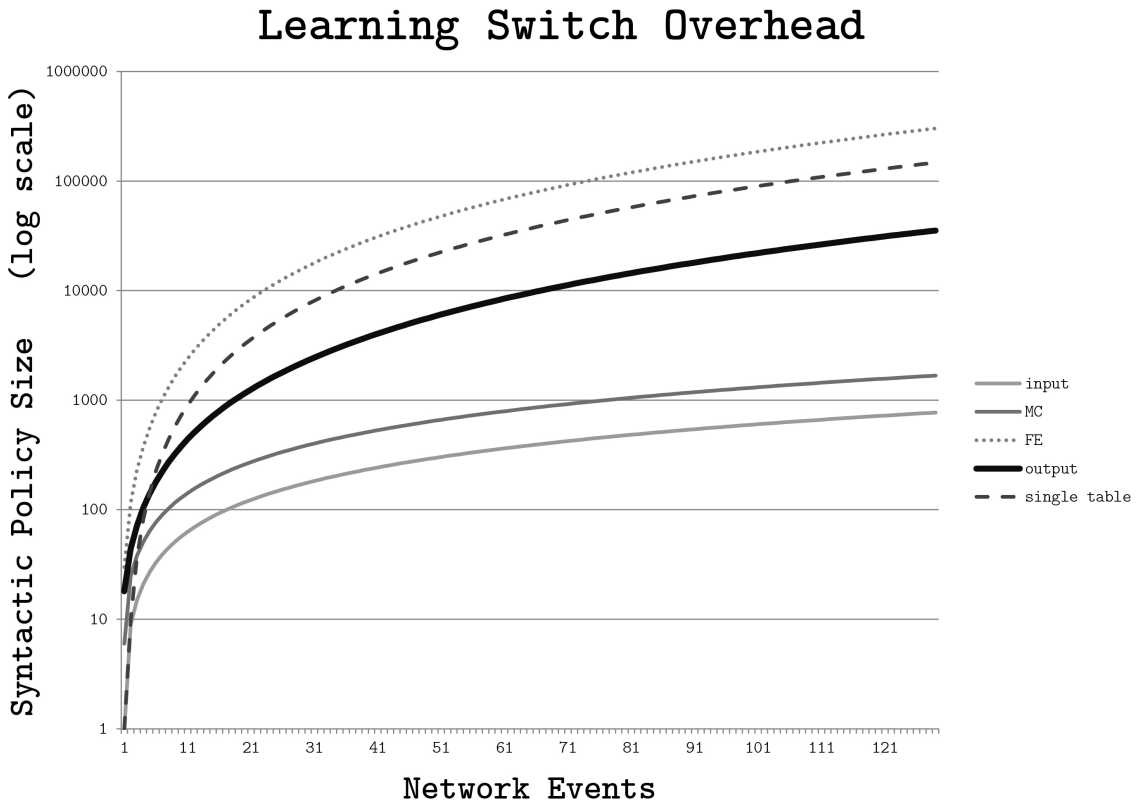

Figure 5.2: The size of compiled virtual pipelines filled with a compiled population-time update. Size measures the number of operators in the policy.

pipeline, traverses the pipeline back-to-front. Because the pipeline is longer, field extraction must consider more interactions between earlier and later stages of the pipeline, yielding a larger extraction stage. However, there is no interaction between different pieces of the egress pipeline produced by multicast consolidation—the crux of that phase lies in ensuring that different packet copies are processed by different pieces of the pipeline. And so the field extraction stage produces a policy much larger than necessary, as evinced by the smallest dashed line in Figure 5.2, labeled "FE" in the legend—the output of field extraction is larger even than single table compilation. But combining the consolidation and extraction stages eliminates the spurious interactions introduced in field extraction and results in a much smaller output policy (labeled "output" in the legend).

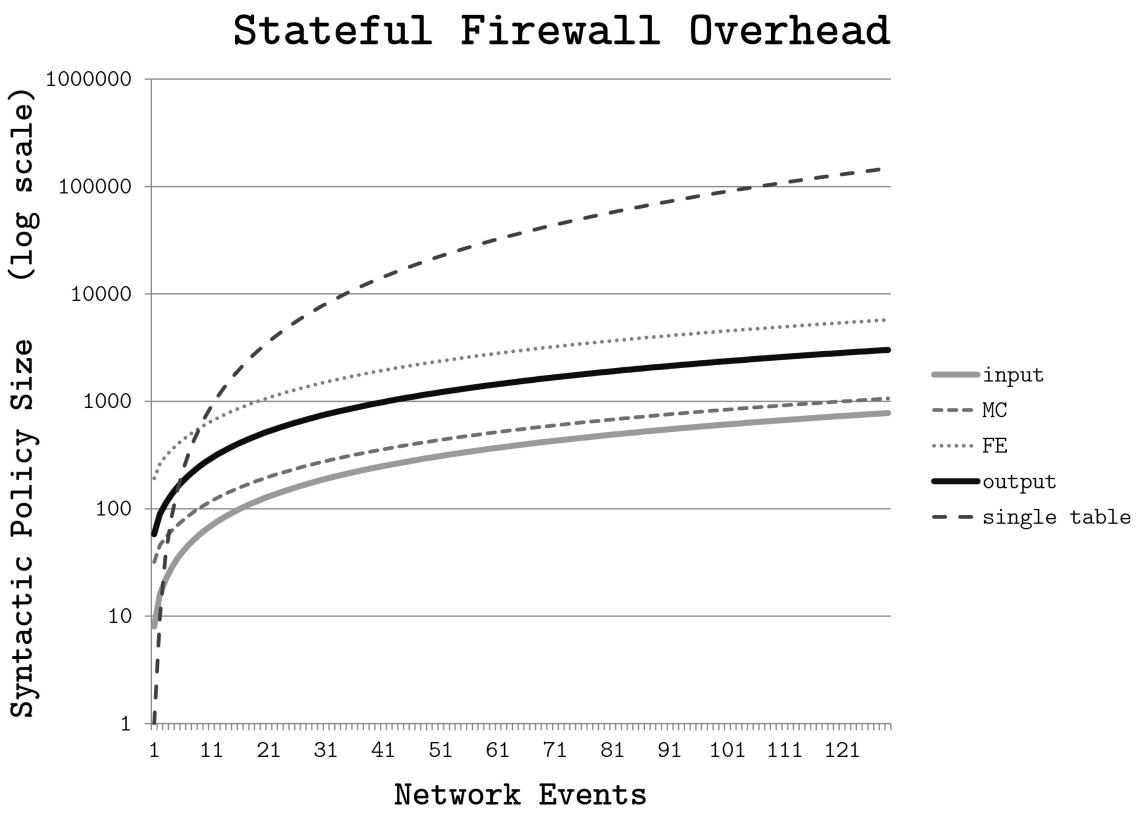

Figure 5.3: The size of compiled virtual pipelines filled with a compiled population-time update. Size measures the number of operators in the policy.

Figure 5.3 shows a similar breakdown for the stateful firewall benchmark. In this case, the size of the field extraction stage is much smaller—still larger than the output policy, but much smaller than the size of single-table compilation. The difference lies in the structure of the firewall table, which drops packets or leaves them unchanged, rather than choosing an output port. As such, field extraction is much simplified, and although the same interference occurs with the multicast consolidation phase, the size of the extraction stage is small enough for the impact to be negligible.

Limitations of the prototype. One might wonder why policy sizes are reported in terms of syntactic size rather than, say, the number of physical tables required to hold them. The current prototype directly implements the algorithms described in this dissertation, with little attempt to promote tail recursion, limit data structure

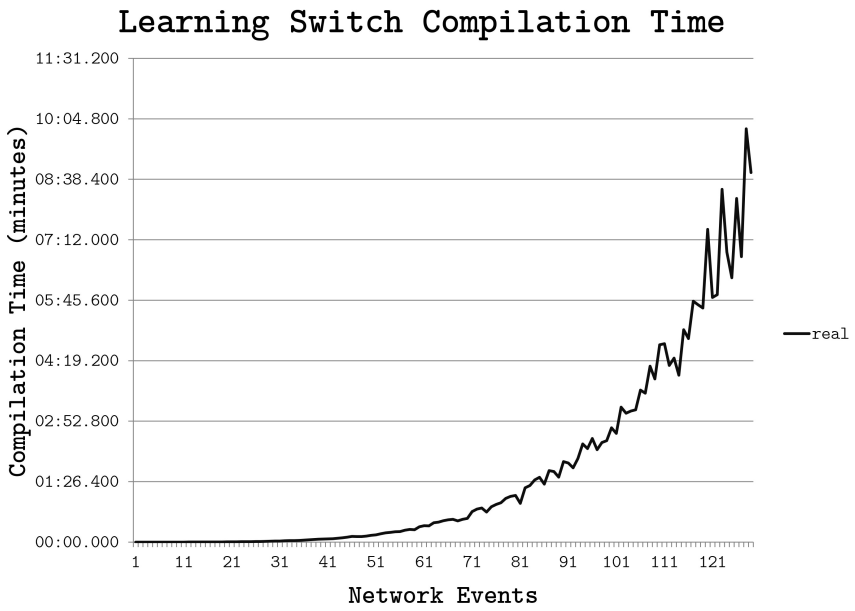

Figure 5.4: The wall-clock time required for compiling a virtual policy and population-time update.

traversals, or curtail the number of objects produced. As such, the OCaml implementation suffers from poor memory performance, running out of memory on examples large enough to fully exercise the physical pipeline, and spending a significant amount of time to compile even smaller examples. Figures 5.4 and 5.5 report the wall-clock time for the pipeline compilation of both benchmarks. Notably, the compilation time of the stateful firewall is much lower than that of the learning switch, which indicates that the problem lies in field extraction and combining the results of field extraction and multicast consolidation.

Figure 5.6[1] shows the memory performance of pipeline compilation on the learning switch benchmark with twenty network events. The X-axis shows the time elapsed since execution began, and the Y-axis shows the amount of memory allocated. The solid regions correspond to cumulative live words in the heap, with the vast majority coming from CNC policy data structures. The dashed line shows all the words allo-

[1]This graph was generated by the TypeRex OCaml Memory Profiler.

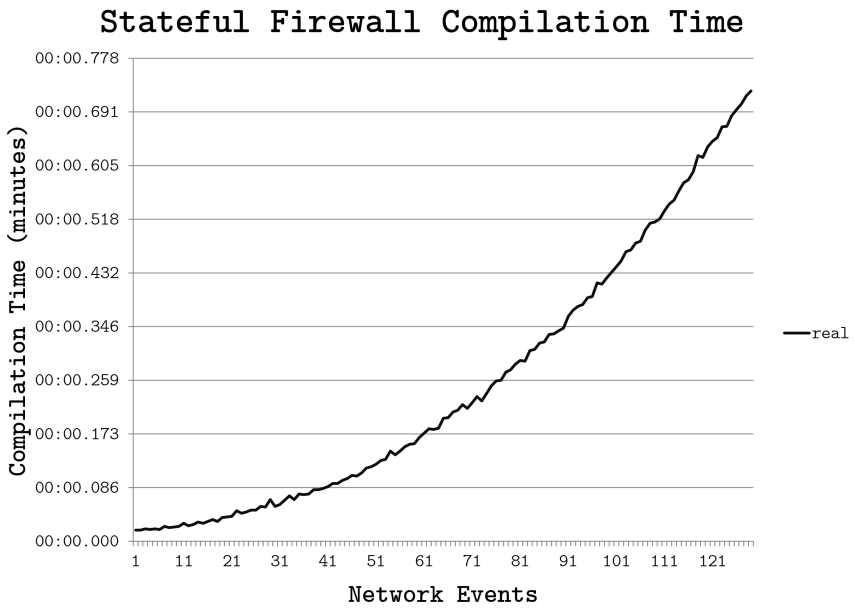

Figure 5.5: The wall-clock time required for compiling a virtual policy and population-time update.

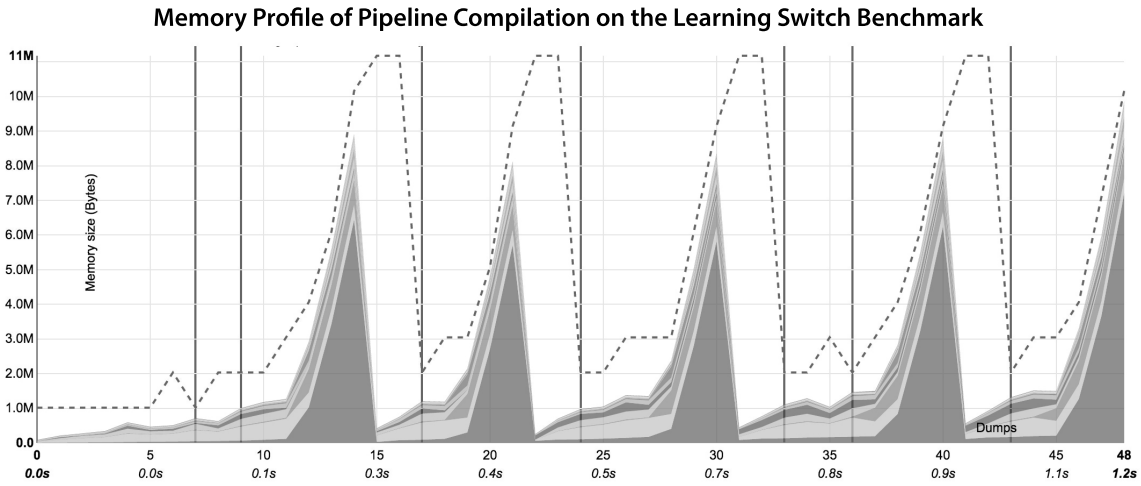

Figure 5.6: The memory performance of pipeline compilation on the learning switch benchmark with twenty network events.

cated on the heap—live or not—and the vertical lines represent compactions by the garbage collector.

There are two things particularly noteworthy about this chart. First, at this number of network events, the size of the input policy is roughly forty rules, corresponding to a policy abstract syntax tree with roughly one hundred twenty nodes, and the size of the output policy is just over twelve hundred nodes. But the amount of live memory repeatedly peaks close to nine megabytes, which is vastly larger than the final size of the policy. Even the intermediate result of the field extraction stage is only on the order of eight thousand nodes. Hence, there is room for substantial improvement in the memory performance of the prototype.

A second point stems from the behavior of memory use over time. In particular, memory use spikes not three times, which one might expect to correspond to the compiler phases, but five times instead. This indicates unnecessary work performed during compilation, such as taking multiple passes over the policy data structure where one would suffice.

Concluding remarks regarding the evaluation. Fortunately, the issues contributing to the performance of the prototype are not insurmountable. The interference between multicast consolidation and field extraction is not fundamental; it can likely be solved by merging the two passes, giving field extraction additional information about feasible paths through the egress pipeline. And we expect that additional memory and run-time profiling will lead to tangible performance improvements in the prototype. The final result then points to compilation as a viable means of automatically deploying higher-level virtual pipelines to switches that support reconfigurable, multi-table pipelines.

Chapter 6

Related Work

The NetKAT and Concurrent NetCore languages, pipeline models, and compilation techniques are inspired by and share some common characteristics with work in both the programming languages and networking communities.

6.1 NetKAT

Kleene algebra is named for its inventor, Stephen Cole Kleene. Much of the basic algebraic theory of KA was developed by John Horton Conway [9]. Kleene algebra with tests was introduced by Kozen [25, 26]. KA and KAT have been successfully applied in many practical verification tasks, including verification of compiler optimizations [28], pointer analysis [36], concurrency control [8], and device drivers [27]. This is the first time KA has been used as a network programming language or applied to verification of networks.

As a programming language, NetKAT is most similar to NetCore [37] and Pyretic [38], which both stemmed from earlier work on Frenetic [11]. NetCore defined the fragment of NetKAT that included parallel composition and Pyretic extended NetCore with sequential composition, although Pyretic did not include a compiler for deploying policies to SDN-enabled switches; rather, Pyretic routed

all traffic through the controller. Neither system defined an equational theory for reasoning about programs, nor did it include Kleene star—unlike these previous languages, NetKAT programs can describe potentially infinite behaviors.

NDLog [32] is a logic programming language with an explicit notion of location and a distributed execution model. In contrast to NDLog, NetKAT and NetCore are designed for programming centralized (not distributed) SDN controllers. Because NDLog is based around Datalog (with general recursion and pragmatic extensions that complicate its semantics), equivalence of NDLog programs is undecidable [47]. NetKAT's Kleene star is able to model network behavior, but has decidable (PSPACE-complete) equivalence.

Several other research groups have proposed domain-specific SDN programming languages [40, 13, 52, 54, 10]. While these network programming languages allow programmers to specify the behavior of each switch using high-level abstractions that a compiler translates to low-level instructions for the underlying hardware, they lack an equational theory, and they do not target switches with reconfigurable pipelines.

6.2 Concurrent NetCore

Concurrent NetCore shares a common core with NetCore [37] and NetKAT [2], but adds table specifications, concurrency, and a type system. These additions require a new approach to the semantics—the denotational techniques used for NetCore and NetKAT do not extend easily to models of concurrency. Moreover, these new features make it possible to express controller requirements as well as next generation switch hardware features. We have focused on specifying the properties of individual switches here, so Kleene star is unnecessary, but it would be interesting to investigate adding it in the future to facilitate reasoning about networks of multi-table switches.

Types. The type system introduced in Concurrent NetCore is akin to a type and effect system in the style of the Tofte-Talpin system [50], with some similarities to the FX language [12] and Deterministic Parallel Java [4]. The types in CNC describe not only the values produced but also characteristics of the computations that produce them—in this case, the fields read and written—and the annotations on the CNC concurrency operator describe an effect: how the fields of a packet should be allocated to the concurrently operating sub-policies. Like Tofte and Talpin, we use the type system to prove that the system behaves at run time. In their TT language, which supports region-based memory management, they prove a correctness property that accounts for memory safety. In our case, we prove that CNC is strongly normalizing and free of race conditions. And, as with the FX language and Deterministic Parallel Java, we rely on effects to characterize the interactions between concurrent operations.

However, CNC is a domain-specific language designed to target network switches. As such, it lacks many features found in the languages associated with other type and effect systems, including functions (higher-order or otherwise), loops, memory allocation, or even the ability to assign expressions to variables (CNC supports assigning constants to fields). Nor do we present an effect inference algorithm for reconstructing the type annotations for concurrent operators, although doing so—in our case—is straightforward.

Concurrency and abstract algebra. Hoare *et al.* define a family of algebras with operators for both sequential and concurrent composition, which they refer to collectively under the common heading *concurrent Kleene algebra* (CKA) [16]. Their development is quite general, with the intention of capturing fundamental properties of program execution that hold even in concurrent architectures with weak memory models, distributed systems with unreliable communication, and in the presence of aggressive program optimizations. Ultimately, they show that CKAs retain suffi-

cient expressive power to encode the sequential and concurrent assertional reasoning techniques of Hoare logic [15] and Jones's rely/guarantee calculus [19].

As such, the technical developments in support of CKAs are quite different from those of NetKAT and Concurrent NetCore. For example, Hoare *et al.* define a trace-based denotational semantics to illustrate four operators: sequential composition, alternation, disjoint parallel composition, and fine-grained concurrent composition. In this semantics, a trace is defined as a *set* of abstract events drawn from a universe EV coupled with a dependence relation $\rightarrow \ \subseteq EV \times EV$ that can be used to induce an event ordering: $e \rightarrow f$ if data or control flow from event e to event f. Trace independence (written $tp \not\leftarrow tq$) indicates that no event in trace tq depends on an event in trace tp. Programs are defined as sets of traces, and the semantics of each operator is defined as follows.

$$
\begin{array}{lll}
\text{concurrent composition} & tp(*)tq & \stackrel{df}{\iff} \ tp \cap tq = \emptyset \\
\text{sequential composition} & tp(;)tq & \stackrel{df}{\iff} \ tp(*)tq \wedge tp \not\leftarrow tq \\
\text{parallel composition} & tp(||)tq & \stackrel{df}{\iff} \ tp(;)tq \wedge tq \not\leftarrow tp \\
\text{alternation} & tp([])tq & \stackrel{df}{\iff} \ tp = \emptyset \vee tq = \emptyset
\end{array}
$$

In contrast, the behavior of a CNC pipeline is described with a small-step operational semantics, which more closely reflects the computational steps of physical switching hardware. That being said, a similar trace-based semantics can likely model the behavior of CNC pipelines, with each trace corresponding to one packet produced by the pipeline. In such a setting, "events" would be field tests and modifications. Concurrent and sequential composition would be defined as in the model of Hoare *et al.*, but CNC union (the + operator) would replace alternation to produce a set of two traces, rather than one or the other. Finally, although CNC does not have an operation that directly corresponds with what Hoare *et al.* call parallel composition, our goal would be to produce a correctness result showing that in well-typed CNC pro-

grams, concurrent composition behaves like their parallel composition, in that there is no potential for fine-grained interleaving with different orderings of dependent events.

As an aside, it is worth noting that the NetKAT/CNC interpretation of "alteration" as "copying union" composition is quite different than that of CKA (as well as other interpretations of Kleene algebra that we are aware of). Typically, "alteration" indicates a form of nondeterminism, wherein one of two behaviors may occur, but not both. "Copying union," on the other hand, implies that *both* behaviors occur simultaneously, but on different copies of a given data packet. Despite the difference, this interpretation still satisfies the KAT equations.

Concurrent Kleene algebra also contains an "exchange" axiom that characterizes the relationship between concurrent and sequential composition as follows.

$$(P * R); (Q * S) \subseteq (P; Q) * (R; S)$$

Here, P, Q, R, and S are programs, and the subset relation compares traces generated by these programs. Intuitively, the program on the right includes more traces, because all of $P; Q$ may interleave with all of $R; S$. The exchange axiom reflects similar reasoning to Lemma 5, which describes the conditions in CNC under which $p; q \equiv p \,||\, q$. Suppose $(p \,||\, r); (q \,||\, s)$ is well typed. Using this lemma, we can see that the following CNC equivalence holds only in certain cases.

$$(p \,||\, r); (q \,||\, s) \equiv (p; q) \,||\, (r; s)$$

In particular, the following additional conditions are necessary. Suppose, as part of our earlier assumption (that the left-hand side of the equivalence is well typed), that

the following sub-derivations are part of that typing derivation.

$$\Gamma \vdash p : (\mathsf{R}_p, \mathsf{W}_p)$$
$$\Gamma \vdash q : (\mathsf{R}_q, \mathsf{W}_q)$$
$$\Gamma \vdash r : (\mathsf{R}_r, \mathsf{W}_r)$$
$$\Gamma \vdash s : (\mathsf{R}_s, \mathsf{W}_s)$$

In that case, the following conditions are needed to show that the equivalence holds.

$$\mathsf{R}_p \cap \mathsf{W}_s = \mathsf{R}_s \cap \mathsf{W}_p = \mathsf{W}_p \cap \mathsf{W}_s = \emptyset$$
$$\mathsf{R}_q \cap \mathsf{W}_r = \mathsf{R}_r \cap \mathsf{W}_q = \mathsf{W}_q \cap \mathsf{W}_r = \emptyset$$

Under these conditions, we can apply Lemma 5 and Lemma 4 (concurrency commutes) to effect the following transformation.

	Assertion	**Reasoning**
	$(p \mid\mid r); (q \mid\mid s)$	
$\equiv$	$p; r; q; s$	Lemma 5.
$\equiv$	$p; (r \mid\mid q); s$	Lemma 5.
$\equiv$	$p; (q \mid\mid r); s$	Lemma 4.
$\equiv$	$p; q; r; s$	Lemma 5.
$\equiv$	$(p; q) \mid\mid (r; s)$	Lemma 5.

The reverse, however, is not the case. That is, if $(p; q) \mid\mid (r; s)$ is well typed, then we can derive $(p \mid\mid r); (q \mid\mid s)$ with no additional constraints.

The remaining algebraic developments surrounding CKA—the encoding of Hoare logic and rely/guarantee-style reasoning—may hold for CNC as well, but further investigation is needed to establish the relationship formally.

Next-generation switch programming languages. Bossart *et al.* [5] propose an architecture for programming "OpenFlow 2.0" switches, which we follow in this dissertation. Bossart's configuration language includes components for programming the packet parser as well as the match-action packet processing. We focus on just

the match-action processing here, but provide a formal semantics and metatheoretic analysis of our work, whereas they provide no semantics. We also consider concurrent and parallel composition, which they do not. Another important inspiration is the ONF's ongoing work on typed table patterns [1].

6.3 Compilation

The single-table compilation algorithm expands on the NetCore compilation algorithm targeting OpenFlow 1.0 tables [37] by adding compilation for sequential composition. Guha *et al.* [13] develop a machine-verified controller, which includes a proved correct implementation of the NetCore compilation algorithm, although they use semantic rather than syntactic proofs.

Just as parts of the CNC language overlap with the P4 language [5], so do parts of pipeline compilation. In particular, the P4 language begins without concurrent or parallel composition, and so has no need for multicast consolidation, nor an immediate need for field extraction; rather, compilation focus on table fitting. Jose *et al.* [20] identify four types of program dependence in P4 programs, akin to the read-after-write, write-after-read, write-after-write, and control dependences standard in the compilers literature. Jose *et al.* use these dependences to specify ordering constraints on virtual tables in an integer linear program, the output of which maps a virtual pipeline into a physical pipeline. While they do not develop a semantics or justify the correctness of the reordering operations, those operations appear similar to the type-based reasoning we present in Chapter 3 for justifying the commutativity and sequentialization of the concurrency operator (Lemmas 4 and 5). Nor do Jose *et. al* attempt to break dependences by setting metadata fields, a technique we employ in both multicast consolidation and field extraction. It would be interesting to combine these approaches in the future. Both the table fitting algorithm of Section 4.2.4 the

integer linear programming approach of Jose *et al.* take exponential time in the worst case, although Jose *et al.* also present greedy approximations. Additional experiments are needed to compare these approaches in practice.

Chapter 7

Summary and Future Work

This dissertation introduces a language for programming networking hardware with reconfigurable packet-processing pipelines. We began by establishing a connection between a packet-processing language and abstract algebra, which gives rise to a sound and complete equational theory. Next, we showed how to extend the language with a type system and support for tables and concurrency, which mirror packet processing pipelines in emerging switch architectures. Using the extended language, we showed how to model the pipelines of three switch architectures, define a virtual pipeline independent of any one physical architecture, and compile from one to the other. We also studied the formal properties of the extended language, establishing a strong normalization result and a sound (but not complete) equational theory, which we used to prove the compilation algorithms correct.

Reconfigurable pipelines are still emerging as a means of flexibly controlling a network of switches, and there are several areas where the results in this dissertation can be extended to expose and exploit more sophisticated hardware features, add more powerful abstractions, and integrate with a more robust, optimization-focused compiler architecture. The remainder of this chapter discusses these avenues for future research.

7.1 Building A Language of Actions

The RMT [6] and FlexPipe [42] architectures both support actions beyond field modification and packet duplication, such as building simple arithmetic expressions over field values and built-in hash functions, assigning expressions to fields, adding and removing fields, and more. Some actions, like packet duplication, are invoked by writing predefined values to particular metadata fields, while others are built into the interfaces for configuring the pipeline, as with field modification. CNC includes operators for both field modification and packet duplication, both of which are given a concrete semantics and play a role in developing the equational theory.

It would be interesting to investigate general properties of actions, with an eye toward extending the CNC language and equational theory with abstract "action" operations. For example, perhaps knowing the fields an action reads and writes will be sufficient to reason about program transformations; this seems reasonable for arithmetic and hash functions. Something like encryption, on the other hand, may require additional reasoning principles to account for the fact that other operations (like arithmetic) cannot be applied to a field after it has been encrypted. Other approaches may include developing a sub-language of actions to embed within CNC, such as the language of arithmetic expressions, or adding first-class language constructs to CNC, such as variable binding and scoping to model adding and removing fields from the packet.

7.2 Adopting Traditional Compilation Techniques

Many features and operations within a switch pipeline bear striking resemblance to aspects of traditional Von Neumann architectures. For example, packets can be thought of as heaps with named locations—`dstip` rather than `0xdeadbeef`—and the pipeline policy is evaluated with respect to each packet; reads and writes to packet

fields act just as reads and writes to the heap. Likewise, metadata fields are akin to registers: the hardware supports a predetermined, fixed set, some of which are subject to special, hardware-specific interpretation.

Compilation techniques from traditional software systems may also apply to the compilation of packet-processing programs. Several compilation techniques in this dissertation introduce the use of metadata fields; one might use techniques drawn from register allocation to minimize the number of metadata fields required. Dead code elimination would certainly be useful, given that the memory in switch pipelines is scarce. And in a setting with Kleene star, loop unrolling might be a reasonable transformation to enable other forms of optimization. It would be interesting to investigate adopting these and other compilation techniques, but also to explore whether a more formal, fundamental connection exists.

7.3 Coordinating Optimizations

Finally, the move toward reconfigurable pipelines in software-defined networks introduces new opportunities for more sophisticated forms of optimization. In this setting, where table space is a scarce commodity, there are three mechanisms one might use to maximize the amount of table space available and minimize the size of policies installed at population time. First, the pipeline configuration itself can optimize the available rule space for a given class of population-time policies, as we saw in Chapter 4 of this dissertation. Second, at population time, the rules destined for a single table can be transformed into semantically equivalent but syntactically fewer rules, in the style of TCAM optimization [31]. Finally, population-time policies can be refactored and split across multiple tables [21, 35]. And from the perspective of rule placement optimizations, there is little difference between a pipeline of tables and a network of switches; in principle, optimizations for one should apply to the other.

In traditional software systems, deciding how best to compose compiler optimizations—called the *phase ordering problem*—is known to be challenging [51, 55, 29, 30]. Some optimizations enable others, while some cause interference, and predicting the interaction is difficult. It would be interesting to consider the interactions between table configuration, single-table optimization, and multi-table optimization, perhaps drawing from the literature on the phase ordering problem to better coordinate enabling CNC policy optimizations.

Other run-time activities also interact with table configuration and rule optimizations. As an example, consider rule caching, where the controller manages a set of population-time policies that are too large to fit in the available pipeline memory by deploying the most popular rules to the pipeline (for low-latency/high-throughput handling) and diverting "missed" packets—those not handled by the cached rules—to the controller (for higher-latency/lower-throughput handling) [22]. Just as some optimizations may enable other optimizations, so too might some optimizations enable more efficient caching algorithms. Other activities, like guaranteeing consistent updates [43] or managing federated network control [18], may benefit from coordinated optimizations as well.

Appendix A

Correctness of the Isolation Algorithm

This section contains the full proofs and supporting lemmas for theorems in Section 2.3.

As Figure A.1 shows, the desugared translation of slice policies relies heavily on predicates on worlds, which we have written $\mathsf{vlan} = w$. For the rest of this section, we will sometimes abbreviate this as simply w. Hence, a policy written $w; p$ is equivalent to $\mathsf{vlan} = w; p$. We also rely on a *denesting* lemma, drawn from [24], that shows how to transform a summation under a star into a series of conjunctions and stars.

Lemma 9 (Denesting)**.**

$$p^*(qp^*)^* \equiv (p+q)^*$$

Proof. Proposition 7 in [24].

$$\begin{aligned}
&(\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0} ::= \\
&\quad \mathsf{let}\ pre = (\mathsf{vlan} = w_0; in; \mathsf{vlan} \leftarrow w + \mathsf{vlan} = w)\ \mathsf{in} \\
&\quad \mathsf{let}\ post = (out; \mathsf{vlan} \leftarrow w_0 + \neg out)\ \mathsf{in} \\
&\quad pre; p; post
\end{aligned}$$

Figure A.1: Slice desugaring.

□

Slice policies and predicates must be tag-free. Intuitively, tag-freedom asserts that policies and predicates neither test nor modify the tag field. Formally, we use the following definitions.

Definition 19 (tag-free Policy)**.** *A policy p is* tag*-free when it commutes with any test of the slice field:*

$$\textit{For all } w,\ \mathsf{tag} = w; p \equiv p; \mathsf{tag} = w.$$

Definition 20 (tag-free Predicate)**.** *A predicate b is* tag*-free when it commutes with any modification of the* tag *field:*

$$\textit{For all } w,\ \mathsf{tag} \leftarrow w; b \equiv b; \mathsf{tag} \leftarrow w.$$

Unsurprisingly, certain commutativity properties also hold on the topology. In particular, the topology may only modify the location information associated with each packet but not the packet itself. Hence, any tests on the packet commute with the topology.

Lemma 10 (Topology Preserves Packets)**.** *For all topologies t and predicates b, if b does not include tests of the form $\mathsf{sw} = v$ for any value v, then $b; t \equiv t; b$.*

Proof. By induction on the structure of t. The base case follows immediately from KA-Seq-Zero. The inductive case is $t = \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + t'$.

	Assertion	**Reasoning**
	$b; (\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + t')$	
$\equiv$	$b; \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + b; t'$	KA-Seq-Dist-L.
$\equiv$	$\mathsf{sw} = sw; \mathsf{pt} = pt; b; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + b; t'$	BA-Seq-Comm.
$\equiv$	$\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; b + b; t'$	KA-Match PA-Mod-Mod-Comm.
$\equiv$	$\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; b + t'; b$	IH.
$\equiv$	$(\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; +t'); b$	KA-Seq-Dist-R.

□

Suppose there exist two slices. The first only emits packets that, after traversing the topology, do not match the ingress predicate of the second. A sequence composed of the first slice, the topology, and the second slice is equivalent to drop.

Lemma 11 (No Slice Sequencing)**.** *For all slice ingress and egress predicates* in_1, out_1, in_2, out_2*, slice identifiers* w_1, w_2*, and policies* s_1, s_2, p, q*, and topologies* t*, such that*

- $s_1 \equiv (\!|\{in_1\}\ w_1 : (p)\ \{out_1\}|\!)^{w_0}$,
- $s_2 \equiv (\!|\{in_2\}\ w_2 : (q)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2$,
- *H1:* p, q, in_1, in_2, out_1, out_2 *are all VLAN-free.*
- *H2:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,

then the following equivalence holds:

$$s_1; t; \mathsf{dup}; s_2 \equiv \mathsf{drop}$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{aligned}
pre_1 &= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\
post_1 &= (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\
s_1 &= pre_1; p; post_1 \\
&= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1)
\end{aligned}$$

$$\begin{aligned}
pre_2 &= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\
post_2 &= (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\
s_2 &= pre_2; q; post_2 \\
&= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2)
\end{aligned}$$

	Assertion	**Reasoning**
	$s_1; t; \mathsf{dup}; s_2$	
$\equiv$	$pre_1; p; post_1; t; \mathsf{dup}; pre_2; q; post_2$	Substitution for s_1 and s_2.
$\equiv$	$pre_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	Substitution for $post_1$.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_1.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + \mathsf{id}); w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	PA-Mod-Filter and KA-Seq-Dist-R.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	BA-Seq-Idem, PA-Mod-Filter, and KA-Seq-Dist-R.
$\equiv$	$pre_1; w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_1.
$\equiv$	$pre_1; p; w_1; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	H1.
$\equiv$	$pre_1; p; w_1; out_1; \mathsf{vlan} \leftarrow w_0; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Dist-L.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; out_1; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H1.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; w_0; out_1; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	PA-Mod-Filter.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; w_0; out_1; t; \mathsf{dup};$ $(w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_2.

$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_0; out_1; t; \mathsf{dup}; w_2);$ $q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Dist-L.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H1, Lemma 10, and PA-Contra.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; in_2; w_0; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	BA-Seq-Comm.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; \mathsf{drop}; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H2.
$\equiv$	$pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Zero, KA-Zero-Seq, and KA-Plus-Zero.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup}; w_1; pre_2; q; post_2$	H1 and Lemma 10.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup}; w_1;$ $(w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$	Substitution for pre_2.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup};$ $(w_1; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_1; w_2); q; post_2$	KA-Seq-Dist-L.
$\equiv$	drop	PA-Contra, KA-Seq-Zero, KA-Zero-Seq, and Plus-Zero. $\square$

Now, suppose there are two slices that neither admit the same packets nor does one eject any packets the other may inject. Every packet that enters the network will either be admitted to the first slice, or the second, or dropped. Intuitively, if the two slices are indeed isolated, then if we restrict the packets that enter the network to only those that will be injected into the first slice, then running both slices together should be equivalent to running the first slice alone.

Theorem 5 (Slice/Slice Isolation)**.** *For all slice ingress and egress predicates* in_1, out_1, in_2, out_2*, slice identifiers* w_1, w_2*, and policies* s_1, s_2, p, q*, and topologies* t*, such that*

- $s_1 \equiv (\!|\{in_1\}\ w_1 : (p)\ \{out_1\}|\!)^{w_0}$,
- $s_2 \equiv (\!|\{in_2\}\ w_2 : (q)\ \{out_2\}|\!)^{w_0}$,

- *H0:* $w_1 \neq w_2$,
- *H1:* p, q, in_1, in_2, out_1, out_2 *are all VLAN-free.*
- *H2:* $in_1; in_2 \equiv \mathsf{drop}$,
- *H3:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,
- *H4:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$,
- *H5:* $out_1; out_2 \equiv \mathsf{drop}$,

then the following equality holds:

$$w_0; in_1; (s_1; t; \mathsf{dup})^* \equiv w_0; in_1; ((s_1 + s_2); t; \mathsf{dup})^*$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{aligned}
pre_1 &= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\
post_1 &= (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\
s_1 &= pre_1; p; post_1 \\
&= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1)
\end{aligned}$$

$$\begin{aligned}
pre_2 &= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\
post_2 &= (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\
s_2 &= pre_2; q; post_2 \\
&= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2)
\end{aligned}$$

Next, note that the following equivalence holds, which we will call L1: $in_1; w_0; s_2 \equiv \mathsf{drop}$.

Assertion	**Reasoning**

L1	$in_1; w_0; s_2$	
	$\equiv in_1; w_0; pre_2; q; post_2$	Substitution for s_2.
	$\equiv in_1; w_0; (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$	Substitution for pre_2.
	$\equiv (in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2); q; post_2$	KA-Seq-Dist-L.
	$\equiv (in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$	PA-Contra and KA-Seq-Zero.
	$\equiv in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2; q; post_2$	KA-Plus-Zero.
	$\equiv in_1; in_2; w_0; w_0; \mathsf{vlan} \leftarrow w_2; q; post_2$	BA-Seq-Comm.
	$\equiv \mathsf{drop}; w_0; w_0; \mathsf{vlan} \leftarrow w_2; q; post_2$	H2.
	$\equiv \mathsf{drop}$	KA-Zero-Seq.

With L1, we can now show that $w_0; in_1; (s_1; t; \mathsf{dup})^* \equiv w_0; in_1; (s_1 + s_2; t; \mathsf{dup})^*$.

	Assertion	**Reasoning**
	$w_0; in_1; ((s_1 + s_2); t; \mathsf{dup})^*$	
$\equiv$	$in_1; w_0; ((s_1 + s_2); t; \mathsf{dup})^*$	BA-Seq-Comm
$\equiv$	$in_1; w_0; (s_1; t + s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 9.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}));$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-R.
$\equiv$	$in_1; w_0; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$ $+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup});$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L, KA-Seq-Dist-R, and KA-Seq-One.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*);$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*)$ $+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*);$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*)$ $+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-R.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup});$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	L1 and KA-Plus-Zero.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup});$ $(\mathsf{id} + (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$ $+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*;$ $(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$ $+in_1; w_0; (s_1; t; \mathsf{dup})^*; \mathsf{drop};$ $t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 11.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$	KA-Seq-Zero and KA-Zero-Seq.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}))$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1; t; \mathsf{dup})^*$	KA-Star-Unroll-R.

$\equiv \quad w_0; in_1; (s_1; t; \mathsf{dup})^*$ | BA-Seq-Comm. $\square$

Shared inbound edges. In the previous section, we showed that two slices with unshared edges are isolated when composed. Next, we relax the restriction on slice edges: given two slices, s_1 and s_2, the ingresses of the two slices may overlap, and so may the egresses. Intuitively, this captures the case where a packet may be copied and processed by both slices simultaneously. Clearly the behavior of the slices, when composed, is not equivalent to one of the slices acting alone; but neither can one slice interfere with the copy of the packet traversing the other slice.

Theorem 6 (Slice/Slice Composition)**.** *For all* tag*-free slice ingress and egress predicates* in_1, out_1, in_2, out_2*, identifiers* w_1, w_2*, policies* s_1, s_2*,* tag*-free policies* p_1, p_2*, and topologies* t*, such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$*,*
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$*,*
- *H0:* $w_1 \neq w_2$*,* $w_1 \neq w_0$*,* $w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$*, and*
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$*, then*

$$((s_1 + s_2); t; \mathsf{dup})^* \equiv (s_1; t; \mathsf{dup})^* + (s_2; t; \mathsf{dup})^*.$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{aligned} pre_1 &= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\ post_1 &= (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\ s_1 &= pre_1; p; post_1 \\ &= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \end{aligned}$$

$$\begin{aligned} pre_2 &= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\ post_2 &= (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\ s_2 &= pre_2; q; post_2 \\ &= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q(out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \end{aligned}$$

	Assertion	**Reasoning**
	$((s_1 + s_2); t; \mathsf{dup})^*$	
$\equiv$	$(s_1; t + s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 9.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*))^*$	KA-Star-Unroll-L.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t + s_2; t; \mathsf{dup}; s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L and KA-Seq-One.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t + \mathsf{drop}; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 11.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup})^*$	KA-Zero-Seq and KA-Plus-Zero.
$\equiv$	$(\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (\mathsf{id} + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*)$	KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$ $+s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-L, KA-Seq-Dist-R. KA-One-Seq, and KA-Seq-One.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$ $+s_1; t; \mathsf{dup}; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; s_1; t; \mathsf{dup});$ $s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Star-Unroll-R.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$ $+s_1; t; \mathsf{dup}; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$ $+(s_1; t; \mathsf{dup})^*; s_1; t; \mathsf{dup}; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R, KA-One-Seq, and KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	Lemma 11, KA-Zero-Seq,

	KA-Seq-Zero and KA-Plus-Zero.
$\equiv \ \mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + \mathsf{id} + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Plus-Idem, KA-Plus-Comm.
$\equiv \ (s_1; t; \mathsf{dup})^* + (s_2; t; \mathsf{dup})^*$	KA-Star-Unroll-L. $\square$

Finally, we show that Corollary 2 follows from Theorem 6.

Corollary 2. *For all* tag*-free slice ingress and egress predicates* in_1, out_1, in_2, out_2, *identifiers* w_1, w_2, *policies* s_1, s_2, tag*-free policies* p_1, p_2, *and topologies* t, *such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$,
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2$, $w_1 \neq w_0$, $w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$,
- *H3:* $in_1; in_2 \equiv \mathsf{drop}$, *then*

$$in_1; \mathsf{tag} = w_0; ((s_1 + s_2); t; \mathsf{dup})^*$$
$$\equiv in_1; \mathsf{tag} = w_0; (s_1; t; \mathsf{dup})^*$$

Proof. By Theorem 6 and KA-Seq-Dist-L, we have $in_1; w_0; (s_1)^* + in_1; w_0; (s_2)^*$.

	Assertion	**Reasoning**
	$in_1; w_0; (s_1)^* + in_1; w_0; (s_2)^*$	
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0 + in_1; w_0; s_2; (s_2)^*$	KA-Star-Unroll-L and KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+in_1; w_0; (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2; (s_2)^*$	Substitution for $\equiv s_2$.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+(in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2);$ $q; post_2; (s_2)^*$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+(in_1; in_2; w_0; w_0; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2);$ $q; post_2; (s_2)^*$	BA-Seq-Comm.

$\equiv\ in_1; w_0; (s_1)^* + in_1; w_0$ $+(\mathsf{drop}; w_0; w_0; \mathsf{vlan} \leftarrow w_2 + in_1; \mathsf{drop});$ $q; post_2; (s_2)^*$	H4 and PA-Contra.
$\equiv\ in_1; w_0; (s_1)^* + in_1; w_0 + \mathsf{drop}$	KA-Seq-Zero, KA-Zero-Seq, and KA-Plus-Zero.
$\equiv\ in_1; w_0; (s_1)^* + in_1; w_0$	KA-Plus-Zero.
$\equiv\ in_1; w_0 + in_1; w_0; s_1; (s_1)^* + in_1; w_0$	KA-Star-Unroll-L.
$\equiv\ in_1; w_0 + in_1; w_0; s_1; (s_1)^*$	KA-Plus-Comm and KA-Plus-Idem.
$\equiv\ in_1; w_0; (s_1)^*$	KA-Star-Unroll-L. $\square$

Weakening the hypotheses. The hypotheses of Theorem 5 restrict its application to two slices running alone on the network. While such a result provides insight into the nature of slice interaction, we show a stronger result in this section that demonstrates slice isolation in the presence of other slices and activity in the network.

In particular, slices drop traffic with any tag not their own or w_0—this prevents them from interfering with traffic that has been injected into another slice. In turn, if a slice is compiled with a tag w, it can run in isolation on the same network as any NetKAT user policy, so long as that policy drops w-tagged traffic.

Definition 21 (Dropping w-tagged Traffic)**.** *A policy p drops w-tagged traffic when*

- $\mathsf{tag} = w; p \equiv \mathsf{drop}$*, and*
- $p; \mathsf{tag} = w \equiv \mathsf{drop}$.

Lemma 12 (No Program Sequencing)**.** *For all slice ingress and egress predicates in, out, slice identifiers w, and policies s, p, q, and topologies t, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$*,*
- *H0: $w \neq w_0$,*
- *H1: p, q, in, and out are tag-free,*
- *H2: $out; t; \mathsf{dup}; q \equiv \mathsf{drop}$ and $q; t; \mathsf{dup}; in \equiv \mathsf{drop}$,*

- *H3: q drops w-tagged traffic,*

then the following equivalences hold:

$$s; t; \mathsf{dup}; q \equiv \mathsf{drop}$$

$$q; t; \mathsf{dup}; s \equiv \mathsf{drop}$$

Proof. First, note that s has the following desugared form.

$$\begin{aligned} pre &= (w_0; in; \mathsf{tag} \leftarrow w + w) \\ post &= (out; \mathsf{tag} \leftarrow w_0 + \neg out) \\ s &= pre; p; post \\ &= (w_0; in; \mathsf{tag} \leftarrow w + w); p; (out; \mathsf{tag} \leftarrow w_0 + \neg out) \end{aligned}$$

Case 1: $s; t; \mathsf{dup}; q \equiv \mathsf{drop}$. We have:

	Assertion	**Reasoning**
	$s; t; \mathsf{dup}; q$	
$\equiv$	$pre; p; post; t; \mathsf{dup}; q$	Substitution for s.
$\equiv$	$pre; p; (out; \mathsf{tag} \leftarrow w_0 + \neg out); t; \mathsf{dup}; q$	Substitution for $post$.
$\equiv$	$pre; p; (out; \mathsf{tag} \leftarrow w_0; t; \mathsf{dup}; q + \neg out; t; \mathsf{dup}; q)$	KA-Seq-Dist-R.
$\equiv$	$pre; p; (\mathsf{tag} \leftarrow w_0; out; t; \mathsf{dup}; q + \neg out; t; \mathsf{dup}; q)$	H1, PA-Mod-Filter-Comm.
$\equiv$	$pre; p; (\mathsf{drop} + \neg out; t; \mathsf{dup}; q)$	H2.
$\equiv$	$pre; p; \neg out; t; \mathsf{dup}; q$	KA-Plus-Comm and KA-Plus-Zero.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + w); p; \neg out; t; \mathsf{dup}; q$	Substitution for pre.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w; w + \mathsf{id}; w); p; \neg out; t; \mathsf{dup}; q$	PA-Mod-Filter and KA-One-Seq.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); w; p; \neg out; t; \mathsf{dup}; q$	KA-Seq-Dist-R.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); p; \neg out; t; \mathsf{dup}; w; q$	H2, Lemma 10.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); p; \neg out; t; \mathsf{dup}; \mathsf{drop}$	H3.
$\equiv$	drop	KA-Seq-Zero.

Case 2: $q; t; \mathsf{dup}; s \equiv \mathsf{drop}$. We have:

Assertion	**Reasoning**

	$q; t; \mathsf{dup}; s$	
$\equiv$	$q; t; \mathsf{dup}; pre; p; post$	Substitution for s.
$\equiv$	$q; t; \mathsf{dup}; (w_0; in; \mathsf{tag} \leftarrow w + w); p; post$	Substitution for pre.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + q; t; \mathsf{dup}; w); p; post$	KA-Seq-Dist-L.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + q; w; t; \mathsf{dup}); p; post$	PA-Obs-Filter-Comm and Lemma 10.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + \mathsf{drop}; t; \mathsf{dup}); p; post$	H3.
$\equiv$	$q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w; p; post$	KA-Zero-Seq and KA-Plus-Zero.
$\equiv$	$q; t; \mathsf{dup}; in; w_0; \mathsf{tag} \leftarrow w; p; post$	BA-Seq-Comm.
$\equiv$	$\mathsf{drop}; w_0; \mathsf{tag} \leftarrow w; p; post$	H2.
$\equiv$	drop	KA-Zero-Seq. $\square$

Proof. First, note that s has the following desugared form.

$$\begin{aligned} pre &= (w_0; in; \mathsf{vlan} \leftarrow w + w) \\ post &= (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \\ s &= pre; p; post \\ &= (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \end{aligned}$$

	Assertion	**Reasoning**
	$((s + q); t; \mathsf{dup})^*$	
$\equiv$	$(s; t + q; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	Lemma 9.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup}; (\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*))^*$	KA-Star-Unroll-L.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup} + q; t; \mathsf{dup}; s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L, KA-Seq-One.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup} + \mathsf{drop}; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	Lemma 12.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup})^*$	KA-Zero-Seq, KA-Plus-Zero.
$\equiv$	$(\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*); (\mathsf{id} + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*)$	KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$ $+s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Seq-Dist-L, KA-Seq-Dist-R, KA-One-Seq, KA-Seq-One.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$ $+s; t; \mathsf{dup}; (\mathsf{id} + (s; t; \mathsf{dup})^*; s; t; \mathsf{dup});$ $q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Star-Unroll-R.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	

	$+s;t;\mathsf{dup};q;t;\mathsf{dup};(q;t;\mathsf{dup})^*$	
	$+(s;t;\mathsf{dup})^*;s;t;\mathsf{dup};q;t;\mathsf{dup};(q;t;\mathsf{dup})^*$	KA-Seq-Dist-R, KA-One-Seq, KA-Star-Unroll-L.
$\equiv$	$\mathsf{id}+s;t;\mathsf{dup};(s;t;\mathsf{dup})^*+q;t;\mathsf{dup};(q;t;\mathsf{dup})^*$	Lemma 12, KA-Zero-Seq, KA-Seq-Zero, KA-Plus-Zero.
$\equiv$	$\mathsf{id}+s;t;\mathsf{dup};(s;t;\mathsf{dup})^*+\mathsf{id}+q;t;\mathsf{dup};(q;t;\mathsf{dup})^*$	KA-Plus-Idem, KA-Plus-Comm.
$\equiv$	$(s;t;\mathsf{dup})^*+(q;t;\mathsf{dup})^*$	KA-Star-Unroll-L. □

A similar corollary holds when the domain of q does not overlap with the ingress of s.

Theorem 7. *Slice Composition*

For all tag*-free slice ingress and egress predicates* in, out*, identifiers* w*, policies* s, q*,* tag*-free policies* p*, and topologies* t*, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- *H0:* $w \neq w_0$,
- *H1:* $out;t;\mathsf{dup};q \equiv \mathsf{drop}$,
- *H2:* $q;t;\mathsf{dup};in \equiv \mathsf{drop}$,
- *H3:* q *drops* w*-tagged traffic, then*

$$((s+q);t;\mathsf{dup})^* \equiv (s;t;\mathsf{dup})^* + (q;t;\mathsf{dup})^*$$

Proof.

	Assertion	**Reasoning**
	$in;\mathsf{tag}=w_0;((s+q);t;\mathsf{dup})^*$	
$\equiv$	$in;\mathsf{tag}=w_0;((s;t;\mathsf{dup})^*+(q;t;\mathsf{dup})^*)$	Theorem 1
$\equiv$	$in;\mathsf{tag}=w_0;(s;t;\mathsf{dup})^*+in;\mathsf{tag}=w_0;(q;t;\mathsf{dup})^*$	KA-Seq-Dist-L
$\equiv$	$in;\mathsf{tag}=w_0;(s;t;\mathsf{dup})^*+$	
	$in;\mathsf{tag}=w_0;(\mathsf{id}+q;t;\mathsf{dup};(q;t;\mathsf{dup})^*)$	KA-Unroll-L

$$
\begin{array}{lll}
\equiv & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* + & \\
& in; \mathsf{tag} = w_0 + & \\
& in; \mathsf{tag} = w_0; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^* & \text{KA-Seq-Dist-L,} \\
& & \text{KA-Seq-One} \\
\equiv & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* + & \\
& in; \mathsf{tag} = w_0 + & \\
& \mathsf{tag} = w_0; in; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^* & in \text{ is tag-free} \\
\equiv & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* + & \\
& in; \mathsf{tag} = w_0 + \mathsf{drop} & \text{H3, KA-Seq-Zero,} \\
& & \text{and KA-Zero-Seq} \\
\equiv & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* + & \\
& in; \mathsf{tag} = w_0; & \text{KA-Plus-Zero} \\
\equiv & in; \mathsf{tag} = w_0; (\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*) + & \\
& in; \mathsf{tag} = w_0; & \text{KA-Unroll-L} \\
\equiv & in; \mathsf{tag} = w_0 + & \\
& in; \mathsf{tag} = w_0; s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + & \\
& in; \mathsf{tag} = w_0; & \text{KA-Seq-Dist-L} \\
& & \text{and KA-Seq-One} \\
\equiv & in; \mathsf{tag} = w_0 + in; \mathsf{tag} = w_0; s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* & \text{KA-Plus-Comm} \\
& & \text{and KA-Plus-Idem} \\
\equiv & in; \mathsf{tag} = w_0; (s; t; \mathsf{dup})^* & \text{KA-Unroll-L} \\
& & \text{and KA-Seq-Dist-L}
\end{array}
$$

□

With this theorem, we can also see that isolation is preserved across n-ary slice composition.

Lemma 13 (Parallel Composition Preserves w-dropping)**.** *If policies p, q drop w-tagged traffic, then $p + q$ drops w-tagged traffic.*

Proof. We have the following hypotheses:

- $\mathsf{tag} = w; p \equiv p; \mathsf{tag} = w \equiv \mathsf{drop}$
- $\mathsf{tag} = w; q \equiv q; \mathsf{tag} = w \equiv \mathsf{drop}$

And we must show the following goals:

- $\mathsf{tag} = w; (p + q) \equiv \mathsf{drop}$
- $(p + q); \mathsf{tag} = w \equiv \mathsf{drop}$

The result is follows from the hypotheses and KA-Seq-Dist-L, KA-Plus-Zero, and the substitution of equals for equals.

□

Lemma 14 (Parallel Composition Preserves Disjoint Boundaries)**.** *If* $p_i; t; \mathsf{dup}; p_j \equiv \mathsf{drop}$ *for all pairs* (i, j) *for* $i \neq j, 1 <= i <= n, 1 <= j <= n$*, then* $p_1; t; \mathsf{dup}; (p_2 + p_3) \equiv \mathsf{drop}$ *and* $(p_2 + p_3); t; \mathsf{dup}; p_1 \equiv \mathsf{drop}$*.*

Proof. The result follows from the hypotheses, KA-Seq-Dist-L, and KA-Seq-Dist-R.

□

Lemma 15 (Slices Drop w-tagged Traffic)**.** *For all slices* $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$ *and tags* w' *such that* $w' \neq w \neq w_0$*,* s *drops* w'*-tagged traffic.*

Proof. First, note that s has the following desugared form.

$$\begin{aligned} pre &= (w_0; in; \mathsf{vlan} \leftarrow w + w) \\ post &= (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \\ s &= pre; p; post \\ &= (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \end{aligned}$$

Goal 1: $\mathsf{tag} = w'; s \equiv \mathsf{drop}$*.* We have:

	Assertion	**Reasoning**
	$\mathsf{tag} = w'; s$	
$\equiv$	$w'; (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	Substitution for s.
$\equiv$	$(w'; w_0; in; \mathsf{vlan} \leftarrow w + w'; w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	KA-Seq-Dist-L.
$\equiv$	$(\mathsf{drop}; in; \mathsf{vlan} \leftarrow w + \mathsf{drop}); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	Hyp. and PA-Contra.
$\equiv$	drop	KA-Seq-Zero, KA-Plus-Zero.

Goal 2: $s; \mathsf{tag} = w' \equiv \mathsf{drop}$*.* We have:

Assertion	Reasoning
$s; \mathsf{tag} = w'$	
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out); w'$	Substitution for s
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0; w' + \neg out; w')$	KA-Seq-Dist-R
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{drop} + \neg out; w')$	Hyp., PA-Mod-Filter, PA-Contra
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; \neg out; w'$	KA-Seq-Zero, KA-Plus-Zero
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); w'; p; \neg out$	Policy/predicates are tag-free
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w; w' + w; w'); p; \neg out$	KA-Seq-Dist-R
$\equiv (w_0; in; \mathsf{drop} + w; w'); p; \neg out$	Hyp., PA-Mod-Filter, PA-Contra
$\equiv (w_0; in; \mathsf{drop} + \mathsf{drop}); p; \neg out$	Hyp., PA-Contra
$\equiv \mathsf{drop}$	KA-Seq-Zero, KA-Plus-Zero.

□

Proposition 1 (N-ary Slices are Isolated)**.** *For slices $s_1, \ldots, s_n$ compiled with tags $w_1, \ldots, w_n$ such that for all pairs (i, j) for $i \neq j, 1 <= i <= n, 1 <= j <= n$,*

- $w_i \neq w_j \neq w_0$*, and*
- $s_i; t; \mathsf{dup}; s_j \equiv s_j; t; \mathsf{dup}; s_i \equiv \mathsf{drop}$*, then*

$$((s_1 + \ldots + s_n); t; \mathsf{dup})^* \equiv (s_1; t; \mathsf{dup})^* + \ldots + (s_n; t; \mathsf{dup})^*$$

Proof. By induction on n. The base case ($n = 1$) is immediate. We proceed with the inductive step.

1. Let $q = (s_1 + \ldots + s_k)$.
2. From Lemma 13 and the hypotheses, we have that q drops w_{k+1}-tagged traffic.
3. From Lemma 14 and the hypotheses, we have that

$$s_{k+1}; t; \mathsf{dup}; q \equiv q; t; \mathsf{dup}; s_{k+1} \equiv \mathsf{drop}$$

.

From Theorem 1 we have:

Assertion	**Reasoning**
$((s_{k+1} + q); t; \mathsf{dup})^* \equiv (s_{k+1}; t; \mathsf{dup})^* + (q; t; \mathsf{dup})^*$	Theorem 1.
$\equiv (s_{k+1}; t; \mathsf{dup})^* + ((s_1 + \ldots + s_k); t; \mathsf{dup})^*$	Substitution for q.
$\equiv (s_{k+1}; t; \mathsf{dup})^* + (s_1; t; \mathsf{dup})^* + \ldots + (s_k; t; \mathsf{dup})^*$	Induction hypothesis.
$\equiv (s_1; t; \mathsf{dup})^* + \ldots + (s_k; t; \mathsf{dup})^* + (s_{k+1}; t; \mathsf{dup})^*$	KA-Plus-Comm. □

Appendix B

Correctness of the Concurrent NetCore Metatheory

This section presents proofs of the lemmas and theorems from Chapter 3.

Lemma 16 (Weakening)**.** *If* $\Gamma_1, \Gamma_2 \vdash p : \tau$ *and* $x \notin p$ *and* $\vdash \tau'$*, then* $\Gamma_1, x : \tau', \Gamma_2 \vdash p : \tau$*.*

Proof. By induction on p. □

Lemma 17 (Substitution)**.** *If* $\Gamma_1, x : \tau', \Gamma_2 \vdash p : \tau$ *and* $\cdot \vdash q : \tau'$*, then* $\Gamma_1, \Gamma_2 \vdash p[q/x] : \tau$*.*

Proof. By induction on p. All cases except for $p = x$ are trivial ($p[q/x] = p$) or by the IH ($p[q/x] = p_1[q/x]; p_2[q/x]$). When $p = x$, we replace the VAR derivation with a weakened form (Lemma 16) of the assumed derivation. □

Lemma 18 (Normalization)**.** *If*

$$\vdash \tau = (\mathsf{R}, \mathsf{W}) \text{ and } \vdash \mathsf{PK} : \tau \text{ and } \cdot \vdash p : \tau$$

then $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ *such that*

1. $\vdash \mathsf{PK}' : \tau$*, and*

2. $\mathsf{PK}' \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

Proof. By induction on the policy p, leaving τ general. The only difficulty is showing that (ConExit) can always successfully merge the results of well typed concurrency, which can be seen by a careful analysis of the cross product, using part (2) of the IH to show that fields not in the write permission W are "read-only". Since the rules are syntax directed, we use rule names below.

(Id) We have $\langle \mathsf{id}, \langle \mathsf{PK}, W \rangle \rangle$ immediately, and we already know that $\vdash \mathsf{PK} : \tau$. By reflexivity, $\mathsf{PK}' \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

(Drop) We have $\langle \mathsf{drop}, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle \mathsf{id}, \langle \emptyset, \mathsf{W} \rangle \rangle$ immediately, and $\vdash \emptyset : \tau$ and $\emptyset \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$ vacuously.

(Match) We take a step to $\langle \mathsf{id}, \{pk \in \mathsf{PK} \mid pk(f) = v\} \rangle$, which is terminal. If $\vdash \mathsf{PK} : \tau$, then $\vdash \{pk \in \mathsf{PK} \mid pk(f) = v\} : \tau$. Since this set of packets is a subset of PK, it is easy to see that $\{pk \in \mathsf{PK} \mid pk(f) = v\} \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

(Not) First, $\langle \neg a, \langle \mathsf{PK}, \mathsf{W} \rangle, \to \rangle \langle a, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle$. By the IH on a with $\cdot \vdash a : \tau$, we know that $\langle a, \mathsf{PK} \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle \rangle$ and $\vdash \mathsf{PK}' : \tau$. So we can then derive

$$\langle a, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}', \mathsf{W} \rangle \rangle \rangle;$$

we conclude by stepping $\langle \mathsf{id}, \langle \mathsf{not}_{\mathsf{PK}}\ \langle \mathsf{PK}', \mathsf{W} \rangle \rangle \rangle \to \langle \mathsf{id}, \langle \mathsf{PK} \setminus \mathsf{PK}', \mathsf{W} \rangle \rangle$, which is terminal. Since $\vdash \mathsf{PK} : \tau$ and $\vdash \mathsf{PK}' : \tau$, we know that $\vdash \mathsf{PK} \setminus \mathsf{PK}' : \tau$.

Finally, for (2), note that $\mathsf{PK} \setminus \mathsf{PK}'$ is a subset of PK, so we are done.

(Modify) We take a step:

$$\langle f \leftarrow v, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle \mathsf{id}, \langle \{pk[f := v] \mid pk \in \mathsf{PK}\}, \mathsf{W} \rangle \rangle.$$

We know that $f \in \mathsf{W}$ and $\vdash \mathsf{PK} : \tau$ guarantees that the packets match their type. The resulting state is terminal, and we have $\vdash \{pk[f := v] \mid pk \in \mathsf{PK}\} : \tau$.

For (2), we can see that $\{pk[f := v] \mid pk \in \mathsf{PK}\} \setminus \mathsf{W}$ is in fact equal to $\mathsf{PK} \setminus \mathsf{W}$.

(Var) Contradictory—can't apply in an empty context.

(Par) We step

$$\langle p + q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to \langle p + q, \langle \mathsf{par}\ \langle \mathsf{PK}, \mathsf{W} \rangle\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle.$$

We have $\cdot \vdash q : \tau_1$ and $\cdot \vdash p : \tau_2$. We can weaken $\vdash \mathsf{PK} : \tau_1 \cup \tau_2$ to find PK well typed at each of the smaller types. Then, by the IH on p and q (with these weakened typing derivations), we know that $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, W \rangle \rangle$ and $\langle q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle \rangle$, (and PK_p and PK_q are well typed at τ). We can therefore find

$$\langle p + q, \langle \mathsf{par}\ \mathsf{PK}\ \mathsf{PK} \rangle \rangle \to^* \langle \mathsf{id} + \mathsf{id}, \langle \mathsf{par}\ \langle \mathsf{PK}_p, \mathsf{W} \rangle\ \langle \mathsf{PK}_q, \mathsf{W} \rangle \rangle \rangle.$$

Note that there are many ways we can take interleave the steps, but we can get *some* result by running all of the steps for p first and then the steps for q. (We will later find that well typed programs are in fact confluent (Lemma 2).) In any case, we can then step

$$\langle \mathsf{id} + \mathsf{id}, \langle \mathsf{par}\ \langle \mathsf{PK}_p, \mathsf{W} \rangle\ \langle \mathsf{PK}_q, \mathsf{W} \rangle \rangle \rangle \to \langle \mathsf{id}, \langle \mathsf{PK}_p \cup \mathsf{PK}_q, \mathsf{W} \rangle \rangle,$$

which is terminal. We can find that these packets are well typed because we took the union of well typed packets.

For (2), we know that $\mathsf{PK}_P \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$ and $\mathsf{PK}_q \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$. We can therefore conclude that $\mathsf{PK}_p \cup \mathsf{PK}_q \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

(Seq) We have $\cdot \vdash p : \tau_1$ and $\cdot \vdash q : \tau_2$. We have $\vdash \mathsf{PK} : \tau_1 \cup \tau_2$; as for parallel composition, we can weaken this typing to find $\vdash \mathsf{PK} : \tau_1$. We can now apply the IH on p (with this weakened derivation), we know that $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W} \rangle \rangle$, where PK_p is well typed. We can therefore find $\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rightarrow^* \langle \mathsf{id}; q, \langle \mathsf{PK}_p, \mathsf{W} \rangle \rangle$. We can then step $\langle \mathsf{id}; q, \langle \mathsf{PK}_p, \mathsf{W} \rangle \rangle \rightarrow \langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle \rangle$. At this point, we know that $\mathsf{PK}_p \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$.

Now, by the IH on q (with a weakened typing derivation for PK'_p), find that $\langle q, \mathsf{PK}'_p \rangle \rightarrow^* \langle \mathsf{id}, \mathsf{PK}_q \rangle$ such that PK_q is well typed—and we have reached a well typed terminal state. Furthermore, $\mathsf{PK}_q \setminus \mathsf{W} \subseteq \mathsf{PK}_p \setminus \mathsf{W} \subseteq \mathsf{PK} \setminus \mathsf{W}$, so we are done.

(Con) We begin by stepping

$$\langle p \ {}_{W_p}||_{W_q} \ q, \langle \mathsf{PK}, \mathsf{W}_p \cup \mathsf{W}_q \rangle \rangle \rightarrow$$
$$\langle p \ {}_{W_p}||_{W_q} \ q, \langle \mathsf{con}_{\mathsf{W}_p \cup \mathsf{W}_q} \ \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle \ \langle \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W}_q \rangle \rangle \rangle,$$

where $\mathsf{W}_p \cap \mathsf{W}_q = \emptyset$ and $\mathsf{R}_p \cap \mathsf{W}_q = \emptyset$ and $\mathsf{W}_p \cap \mathsf{R}_q = \emptyset$.

We have $\vdash \mathsf{PK} : (\mathsf{R}_p, \mathsf{W}_p) \cup (\mathsf{R}_q, \mathsf{W}_q)$. Note that $(\mathsf{R}_p, \mathsf{W}_p) \cup (\mathsf{R}_q, \mathsf{W}_q) = (\mathsf{R}_P \cup \mathsf{R}_q, \mathsf{W}_p \cup \mathsf{W}_q)$ because of the disjointness conditions in the premise of the typing rule. We can therefore weaken the packet typing $\cdot \vdash \mathsf{PK} : (\mathsf{R}_p \cup \mathsf{R}_q, \mathsf{W}_p \cup \mathsf{W}_q)$ to find $\cdot \vdash \mathsf{PK} \setminus W_q : (\mathsf{R}_p \cup \mathsf{R}_q, \mathsf{W}_p)$ and $\cdot \vdash \mathsf{PK} \setminus W_p : (\mathsf{R}_q \cup \mathsf{R}_q, \mathsf{W}_q)$. With these packet typings, we can now apply the IH, finding that

$$\langle p, \langle \mathsf{PK} \setminus W_q, \mathsf{W}_p \rangle \rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W}_p \rangle \rangle$$

and

$$\langle q, \langle \mathsf{PK} \setminus W_p, \mathsf{W}_q \rangle \rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W}_q \rangle \rangle.$$

Applying the congruence rules to these derivations, we eventually reach the state $\langle \mathsf{id}\ {}_{W_p}||_{W_q}\ \mathsf{id}, \langle \mathsf{con}_{\mathsf{W}_p \cup \mathsf{W}_q}\ \langle \mathsf{PK}_p, \mathsf{W}_p \rangle\ \langle \mathsf{PK}_q, \mathsf{W}_q \rangle \rangle \rangle$, From there we step to $\langle \mathsf{id}, \mathsf{PK}_p \times \mathsf{PK}_q \rangle$

Finally, we must show that (a) $\mathsf{PK}_p \times \mathsf{PK}_q$ is well defined, and (b) that $\mathsf{PK}_p \times \mathsf{PK}_q \setminus \mathsf{W}_p \cup \mathsf{W}_q \subseteq \mathsf{PK} \setminus \mathsf{W}_p \cup \mathsf{W}_q$.

For the cross product to be well defined, it must be the case that for all pairs of packets $pk_p \in \mathsf{PK}_p$ and $pk_q \in \mathsf{PK}_q$, every field $f \in \mathsf{F}$, one of the following must adhere:

- $pk_p(f)$ is defined and $pk_q(f) = \bot$; or
- $pk_q(f)$ is defined and $pk_p(f) = \bot$; or
- $pk_p(f) = pk_q(f)$ (or both are undefined); or
- there is a conflict in a common field and $pk_p \times pk_q$ is undefined.

We will show that those fields in W_p fall into the first class, those fields in W_q fall into the second, and the rest fall into the last two. By the IH for p, we know that $\mathsf{PK}_p \setminus \mathsf{W}_p \subseteq (\mathsf{PK} \setminus \mathsf{W}_q) \setminus \mathsf{W}_p$ and, likewise, $\mathsf{PK}_q \setminus \mathsf{W}_q \subseteq (\mathsf{PK} \setminus \mathsf{W}_p) \setminus \mathsf{W}_q$. The writable fields in W_p of packets in PK_p aren't found at all in the packets in PK_q. We know that $\mathsf{PK}_q \setminus \mathsf{W}_q \subseteq (\mathsf{PK} \setminus \mathsf{W}_p) \setminus \mathsf{W}_q$, and $f \in \mathsf{W}_p$ implies $f \notin \mathsf{W}_q$; it would therefore be a contradiction to have a packet with a field $f \in \mathsf{W}_P$ in PK_q, since such a packet cannot exist in $(\mathsf{PK} \setminus \mathsf{W}_p) \setminus \mathsf{W}_q$. Similarly, the writable fields in W_q of packets in PK_q aren't found at all in the packets in PK_p. Finally, we can rewrite the right-hand sides of the (2) parts of the IH to see that PK_p and PK_q are bounded by $\mathsf{PK} \setminus \mathsf{W}_p \cup \mathsf{W}_q$. That is, all of the read-only packets in PK_p and PK_q haven't changed their fields at all from PK.

For (b), suppose $pk \in (\mathsf{PK}_p \times \mathsf{PK}_q) \setminus \mathsf{W}_p \cup \mathsf{W}_q$. We must show that $pk \in \mathsf{PK} \setminus \mathsf{W}_p \cup \mathsf{W}_q$. As we have seen for (a), the fields in $\mathsf{Dom}\,(pk)$ are precisely

those which weren't written by p or q—the third case. So there exist $pk_p \in \mathsf{PK}_p$ and $pk_q \in \mathsf{PK}_q$ such that for all fields in $\mathsf{F} \setminus (\mathsf{W}_p \cup \mathsf{W}_q)$, we know that $pk(f) = pk_p(f) = pk_q(f)$. We can therefore conclude that:

$$\begin{aligned}(\mathsf{PK}_p \times \mathsf{PK}_q) \setminus (\mathsf{W}_p \cup \mathsf{W}_q) &= \mathsf{PK}_p \setminus (\mathsf{W}_p \cup \mathsf{W}_q) \\ &= \mathsf{PK}_p \setminus (\mathsf{W}_p \cup \mathsf{W}_q) \\ &\subseteq \mathsf{PK} \setminus (\mathsf{W}_p \cup \mathsf{W}_q).\end{aligned}$$

□

Lemma 19 (Diamond property)**.** *If $\langle p, \delta\rangle \to \sigma_1$ and $\langle p, \delta\rangle \to \sigma_2$ (such that $\sigma_1 \neq \sigma_2$) then there exists a σ' such that $\sigma_1 \to \sigma'$ and $\sigma_2 \to \sigma'$.*

Proof. By induction on p, with cases on the step taken.

($p = \mathsf{id}$) Contradictory—doesn't step.

(Drop) There is only one way to step, so $\sigma_1 = \sigma_2$.

(Match) Can only step in one way, so $\sigma_1 = \sigma_2$.

(Modify) Can only step in one way, so $\sigma_1 = \sigma_2$.

(ParEnter) We step from $\delta = \langle \mathsf{PK}, \mathsf{W}\rangle$ to $\langle \mathsf{par}\ \delta\ \delta\rangle$. There are no other possible steps, so $\sigma_1 = \sigma_2$.

(ParL) We have $p = p_1 + p_2$. There are two cases: the other step is also by ParL (and we use the IH on p_1 and ParL to join back up), or the other step is by ParR.

We have $\delta = \langle \mathsf{par}\ \delta_1\ \delta_2\rangle$. In one derivation, we apply the congruent step: $\langle p_1, \delta_1\rangle \to \langle p_1', \delta_1'\rangle$, yielding an outer state of $\langle p_1' + p_2, \delta_1' + \delta_2\rangle$; in the other derivation, we step $\langle p_2, \delta_2\rangle \to \langle p_2', \delta_2'\rangle$, yielding an outer state of $\langle p_1 + p_2', \delta_1 + \delta_2'\rangle$. Each side can then use the other's premise to step to $\langle p_1' + p_2', \delta_1' + \delta_2'\rangle$, and we are done.

(ParR) As for ParL.

(ParExit) We step from $\delta = \langle \mathsf{par}\ \langle \mathsf{PK}_{p_1}, \mathsf{W} \rangle\ \langle \mathsf{PK}_{p_2}, \mathsf{W} \rangle \rangle$ to $\delta' = \langle \mathsf{PK}_{p_1} \cup \mathsf{PK}_{p_2}, \mathsf{W} \rangle$. Since p_1 and p_2 are terminal, no other step can be taken, and $\sigma_1 = \sigma_2 = \langle \mathsf{id}, \delta' \rangle$.

(SeqEnter) There is only one step to take, so $\sigma_1 = \sigma_2$.

(SeqL) We have $p = p_1; p_2$, and p_1 takes two different steps. We can join up the executions by the IH on p_1 and SeqL.

(SeqR) There is only one step to take, so $\sigma_1 = \sigma_2$.

(NotEnter) There is only one way to step, so $\sigma_1 = \sigma_2$.

(NotInner) We have $p = a$ and a takes two different steps. By the IH on a and NotInner, we can rejoin the executions.

(NotExit) Contradictory—there is only one way to step.

(ConEnter) There is only one way to step, so $\sigma_1 = \sigma_2$.

(ConL) We step to $\langle p'\ \mathsf{w}_p||_{\mathsf{w}_q}\ q, \langle \mathsf{con}_\mathsf{W}\ \delta'_p\ \delta_q \rangle \rangle$, since $\langle p, \delta_p \rangle \to \langle p', \delta'_p \rangle$. If the other step is also by ConL, then we can just apply the IH.

If the other step is by ConR, we have $\langle q, \delta_q \rangle \to \langle q', \delta'_q \rangle$, and the whole term stepping to $\langle p\ \mathsf{w}_p||_{\mathsf{w}_q}\ q', \langle \mathsf{con}_\mathsf{W}\ \delta_p\ \delta'_q \rangle \rangle$. We can step this term by applying ConL with our premise about p stepping above. Similarly, we can step our term above via ConR with the premise here. The terms coincide, stepping to $\langle p'\ \mathsf{w}_p||_{\mathsf{w}_q}\ q', \langle \mathsf{con}_\mathsf{W}\ \delta'_p\ \delta'_q \rangle \rangle$.

(ConR) As for ConL.

(ConExit) There is only one way to step, so $\sigma_1 = \sigma_2$.

□

Lemma 20 (Confluence)**.** *If $\sigma \to^* \sigma_1$ and $\sigma \to^* \sigma_2$ then there exists σ' such that $\sigma_1 \to^* \sigma'$ and $\sigma_2 \to^* \sigma'$.*

Proof. By induction on the derivation of $\sigma \to^* \sigma_1$, using the diamond property (Lemma 19).

□

Lemma 21 (Predicates are singular)**.** *For all packets pk, $[\![a]\!]\, pk \subseteq \{pk\}$.*

Proof. By induction on a.

($a = \mathsf{id}$) $[\![a]\!]\, pk = \{pk\}$.

($a = \mathsf{drop}$) $[\![a]\!]\, pk = \emptyset$.

($a = f = v$) $[\![f = v]\!]$ is either $\{pk\}$ or $\emptyset$, depending on the value of $pk(f)$.

($a = \neg a'$) By the IH, we know that $[\![a']\!]\, pk$ is either $\{pk\}$ or $\emptyset$, so $\{pk\} \setminus ([\![a']\!]\, pk)$ is also either $\{pk\}$ or $\emptyset$—whichever $[\![a']\!]\, pk$ was not.

($a = a_1 + a_2$) By the IH, $[\![a_1]\!]\, pk \subseteq \{pk\}$ and $[\![a_2]\!]\, pk \subseteq \{pk\}$, so $[\![a_1]\!]\, pk \cup [\![a_2]\!]\, pk \subseteq \{pk\}$.

($a = a_1; a_2$) By the IH, $[\![a_1]\!]\, pk \subseteq \{pk\}$. If $[\![a_1]\!] = \emptyset$, then $\bigcup_{pk \in \emptyset} [\![a_2]\!]\, pk = \emptyset \subseteq \{pk\}$ immediately. If, on the other hand, $[\![a_1]\!] = \{pk\}$, then:

$$\begin{aligned} [\![a_1; a_2]\!]\, pk &= \textstyle\bigcup_{pk' \in [\![a_1]\!] pk} [\![a_2]\!]\, pk' \\ &= [\![a_2]\!]\, pk && \text{(because } [\![a_1]\!]\, pk = \{pk\}\text{)} \\ &\subseteq \{pk\} && \text{(by the IH)} \end{aligned}$$

and we are done.

□

Lemma 22 (Adequacy)**.** *If* $\vdash \tau = (\mathsf{R}, \mathsf{W})$ *and* $\cdot \vdash p : \tau = (\mathsf{R}, \mathsf{W})$ *with no concurrency, then for all packets* $\vdash \mathsf{PK} : \tau$*,* $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ *and* $\mathsf{PK}' = \bigcup_{pk \in \mathsf{PK}} [\![p]\!]\, pk$*, where:*

$$
\begin{aligned}
[\![p]\!] &\in \mathsf{PK} \to \mathcal{P}(\mathsf{PK}) \\
[\![\mathsf{id}]\!]\, pk &= \{pk\} \\
[\![\mathsf{drop}]\!]\, pk &= \emptyset \\
[\![f = v]\!]\, pk &= \begin{cases} \{pk\} & pk(f) = v \\ \emptyset & \textit{otherwise} \end{cases} \\
[\![f \leftarrow v]\!]\, pk &= \{pk[f := v]\} \\
[\![\neg a]\!]\, pk &= \{pk\} \setminus ([\![a]\!]\, pk) \\
[\![p + q]\!]\, pk &= [\![p]\!]\, pk \cup [\![q]\!]\, pk \\
[\![p; q]\!]\, pk &= \bigcup_{pk' \in [\![p]\!] pk} [\![q]\!]\, pk'
\end{aligned}
$$

Proof. By induction on $\cdot \vdash p : \tau$, noting that PK' always exists by the strong normalization result (Lemmas 1 and 2).

(Id) $\mathsf{PK}' = \mathsf{PK} = \bigcup_{pk \in \mathsf{PK}} pk = \bigcup_{pk \in \mathsf{PK}} [\![\mathsf{id}]\!]\, pk$.

(Drop) $\mathsf{PK}' = \emptyset = \bigcup_{pk \in \mathsf{PK}} \emptyset = \bigcup_{pk \in \mathsf{PK}} [\![\mathsf{drop}]\!]\, pk$.

(Match) $\mathsf{PK}' = \{pk \in \mathsf{PK} \mid pk(f) = v\} = \bigcup_{pk \in \mathsf{PK}} [\![f = v]\!]\, pk$.

(Modify) $\mathsf{PK}' = \{pk[f := v] \mid pk \in \mathsf{PK}\} = \bigcup_{pk \in \mathsf{PK}} [\![f \leftarrow v]\!]\, pk$.

(Not) By the IH, $\langle a, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ such that $\mathsf{PK}' = \bigcup_{pk \in \mathsf{PK}} [\![a]\!]\, pk$. We can then see that $\langle \neg a, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK} \setminus \mathsf{PK}', \mathsf{W} \rangle\rangle$, where $\mathsf{PK} \setminus \mathsf{PK}' = (\bigcup_{pk \in \mathsf{PK}} \{pk\}) \setminus (\bigcup_{pk \in \mathsf{PK}} [\![a]\!]\, pk)$. We must show that this is equivalent to $\mathsf{PK}'' = \bigcup_{pk \in \mathsf{PK}} [\![\neg a]\!]\, pk = \bigcup_{pk \in \mathsf{PK}} (\{pk\} \setminus [\![a]\!]\, pk)$.

Suppose $pk \in \mathsf{PK}''$. Then $pk \in \{pk\}$ but $pk \notin [\![a]\!]\, pk$. Since predicates are singular (Lemma 21), we know that $pk \notin [\![a]\!]\, pk'$ for any $pk' \neq pk$. So then $pk \in (\bigcup_{pk' \in \mathsf{PK}} \{pk'\}) \setminus (\bigcup_{pk' \in \mathsf{PK}} [\![a]\!]\, pk') = \mathsf{PK} \setminus \mathsf{PK}'$.

On the other hand, suppose $pk \in \mathsf{PK} \setminus \mathsf{PK}'$. This means that $pk \in \mathsf{PK}$ but $pk \notin \mathsf{PK}' = \bigcup_{pk' \in \mathsf{PK}} [\![a]\!]\, pk'$. This means that, in particular, $pk \notin [\![a]\!]\, pk$, so then $pk \in \{pk\} \setminus [\![a]\!]\, pk$, so $pk \in \bigcup_{pk' \in \mathsf{PK}} (\{pk'\} \setminus [\![a]\!]\, pk') = \mathsf{PK}''$.

(Var) Contradictory—we assumed that p was well typed in an empty context.

(Par) We have $\langle p + q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$. From this derivation, we can extract the derivations $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle$ and $\langle q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle\rangle$ such that $\mathsf{PK}' = \mathsf{PK}_p \cup \mathsf{PK}_q$.

By the IHs on p and q, we know that $\bigcup_{pk \in \mathsf{PK}} [\![p]\!]\, pk = \mathsf{PK}_p$ and $\bigcup_{pk \in \mathsf{PK}} [\![q]\!] = \mathsf{PK}_q$, respectively. It is then straightforward to see that $\bigcup_{pk \in \mathsf{PK}} [\![p]\!]\, pk \cup [\![q]\!]\, pk = \bigcup_{pk \in \mathsf{PK}} [\![p]\!] \cup \bigcup_{pk \in \mathsf{PK}} [\![q]\!]$.

(Seq) We have $\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$. We can extract the derivations $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle$ and $\langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$.

By the IH on p, we know that $\mathsf{PK}_p = \bigcup_{pk \in \mathsf{PK}} [\![p]\!]\, pk$. Then, by the IH on q, we know that:

$$\begin{aligned}
\mathsf{PK}' &= \bigcup_{pk' \in \mathsf{PK}_p} [\![q]\!]\, pk' \\
&= \bigcup_{pk' \in \bigcup_{pk \in \mathsf{PK}} [\![p]\!] pk} [\![q]\!]\, pk' \\
&= \bigcup_{pk \in \mathsf{PK}} \bigcup_{pk' \in [\![p]\!] pk} [\![q]\!]\, pk' \\
&= \bigcup_{pk \in \mathsf{PK}} [\![p; q]\!]\, pk
\end{aligned}$$

which concludes this case.

(Con) Contradictory—we assumed there was no concurrency.

□

Lemma 23 (Packets aren't ever created)**.** *If* $\langle p, \langle \emptyset, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle$ *then* $\mathsf{PK} = \emptyset$.

Proof. By induction on p.

- ($p = \mathsf{id}$) Immediate.
- ($p = \mathsf{drop}$) By Drop.
- ($p = p + q$) By the IHs, p and q both produce $\emptyset$, so their union is also $\emptyset$.
- ($p = p; q$) By the first IH, p produces $\emptyset$; by the second IH, running q on $\emptyset$ also produces $\emptyset$.
- ($p = \neg a$) By the IH on a and the fact that $\emptyset \setminus \emptyset = \emptyset$.
- ($p = p \ {}_{\mathsf{W}_p}||_{\mathsf{W}_q} \ q$) We know that $\emptyset \setminus \mathsf{W}_p = \emptyset \setminus \mathsf{W}_q = \emptyset$, so we know that p and q separately produce $\emptyset$. We then know that $\emptyset \times \emptyset = \emptyset$.

□

Lemma 24 (Monotonicity of policies)**.** *If* $\langle p, \langle \mathsf{PK}_1, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_1, \mathsf{W} \rangle\rangle$ *and* $\mathsf{PK}_2 \subseteq \mathsf{PK}_1$ *then* $\langle p, \langle \mathsf{PK}_2, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_2, \mathsf{W} \rangle\rangle$ *such that* $\mathsf{PK}'_2 \subseteq \mathsf{PK}'_1$.

Proof. By induction on p.

- ($p = \mathsf{id}$) Immediate.
- ($p = \mathsf{drop}$) By Drop—both versions return $\emptyset$.
- ($p = p + q$) By the IHs, p and q both produce subsets, and the union of subsets is a subset of a union of supersets.
- ($p = p; q$) By the first IH, p produces a subset; by the second IH, running q on a subset also produces a subset.
- ($p = \neg a$) We know by the IH that a on PK_1 produces PK'_1 implies that a on PK_2 produces PK'_2 such that $\mathsf{PK}'_2 \subseteq \mathsf{PK}'_1$.

Suppose $pk \in \mathsf{PK}_2$, so $pk \in \mathsf{PK}_1$. Since a is necessarily concurrency free—there are no concurrent prodicates—by Lemma 3, it suffices to show that $\bigcup_{pk \in \mathsf{PK}_2} [\![\neg a]\!]\, pk \subseteq \bigcup_{pk \in \mathsf{PK}_2} [\![\neg a]\!]\, pk$.

$$\begin{aligned}
\mathsf{PK}_2' &= \bigcup_{pk \in \mathsf{PK}_2} [\![\neg a]\!]\, pk \\
&= \bigcup_{pk \in \mathsf{PK}_2} \{pk\} \setminus [\![a]\!]\, pk \\
&\subseteq \bigcup_{pk \in \mathsf{PK}_1} \{pk\} \setminus [\![a]\!]\, pk \\
&= \bigcup_{pk \in \mathsf{PK}_1} [\![\neg a]\!]\, pk \\
&= \mathsf{PK}_1'
\end{aligned}$$

The key observation is that $[\![\neg a]\!]$ operates pointwise on packets—packets that are rejected when checking PK_1 are also rejected when checking PK_2.

- ($p{=}p\ {}_{\mathsf{W}_p}||_{\mathsf{W}_q}\ q$) We know that each branch produces a subset, and that the $\times$ on subsets is a subset of a $\times$ of supersets.

□

Lemma 25 (Times distributes over union.)**.** $\mathsf{PK} \times (\mathsf{PK}_1 \cup \mathsf{PK}_2) = \mathsf{PK} \times \mathsf{PK}_1 \cup \mathsf{PK} \times \mathsf{PK}_2$.

Proof. The $\times$ operator is defined pointwise on packets in its operand sets. Hence:

$$\begin{aligned}
& \mathsf{PK} \times (\mathsf{PK}_1 \cup \mathsf{PK}_2) \\
= \ & \{\, pk_1 \times pk_2 |\, pk_1 \in \mathsf{PK}, pk_2 \in (\mathsf{PK}_1 \cup \mathsf{PK}_2)\} \\
= \ & \{\, pk_1 \times pk_2 |\, pk_1 \in \mathsf{PK}, pk_2 \in \mathsf{PK}_1\} \cup \{\, pk_1 \times pk_2 |\, pk_1 \in \mathsf{PK}, pk_2 \in \mathsf{PK}_2\} \\
= \ & \mathsf{PK} \times \mathsf{PK}_1 \cup \mathsf{PK} \times \mathsf{PK}_2
\end{aligned}$$

□

Lemma 26 (Times is commutative.)**.** $\mathsf{PK}_1 \times \mathsf{PK}_2 = \mathsf{PK}_2 \times \mathsf{PK}_1$.

Proof. Immediate from the definition of times ($\times$).

□

Lemma 27 (Times is associative.)**.** $(\mathsf{PK}_1 \times \mathsf{PK}_2) \times \mathsf{PK}_3 = \mathsf{PK}_1 \times (\mathsf{PK}_2 \times \mathsf{PK}_3)$.

Proof. Immediate from the definition of times ($\times$).

□

Lemma 28 (Packet extension)**.** *If* $\vdash p : (\mathsf{R}_p, \mathsf{W}_p)$ *and* $\vdash \mathsf{PK} : (\mathsf{R}_p, \mathsf{W}_p)$ *and* $\mathsf{R}_p \cap \mathsf{W}_q = \mathsf{R}_q \cap \mathsf{W}_p = \mathsf{W}_p \cap \mathsf{W}_q = \emptyset$ *then* $\langle p, \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W}_p \rangle\rangle$ *and* $\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W}_p \rangle\rangle$.

Proof. By induction on the typing derivation.

(Id) Immediate.

(Drop) Immediate, stepping by Drop, we have $\emptyset = \emptyset \times \mathsf{PK} \setminus \mathsf{W}_p$.

(Match) Immediate, stepping by Match. We have $\{pk \in \mathsf{PK} \setminus \mathsf{W}_q \mid pk(f) = v\} = \{pk \in \mathsf{PK} \mid pk(f) = v\}$, since $(\mathsf{R}_p \cup \mathsf{W}_p) \cap \mathsf{W}_q = \emptyset$ and $f \in \mathsf{R}_p \cup \mathsf{W}_p$. Since we haven't altered any packets, we know that $\{pk \in \mathsf{PK} \mid pk(f) = v\} = \{pk \in \mathsf{PK} \mid pk(f) = v\} \times \mathsf{PK} \setminus \mathsf{W}_p$, and we are done.

(Not) By the IH, a on $\mathsf{PK} \setminus \mathsf{W}_q$ produces PK_a and a on PK produces $\mathsf{PK}_a \times \mathsf{PK} \setminus \mathsf{W}_p$.

We must show that $\neg a$ on $\mathsf{PK} \setminus \mathsf{W}_q$ produces some PK_p and $\neg a$ on PK produces $\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p$. We find that the first case produces $\mathsf{PK} \setminus \mathsf{W}_q \setminus \mathsf{PK}_a$, while the latter produces $\mathsf{PK} \setminus (\mathsf{PK}_a \times \mathsf{PK} \setminus \mathsf{W}_p)$. We must show that $\mathsf{PK} \setminus (\mathsf{PK}_a \times \mathsf{PK} \setminus \mathsf{W}_p) = (\mathsf{PK} \setminus \mathsf{W}_q \setminus \mathsf{PK}_a) \times \mathsf{PK} \setminus W_p$. We can find this directly from the fact that $(\mathsf{R}_p \cup \mathsf{W}_p) \cap \mathsf{W}_q$: we will always be in the "both functions defined" cases of $\times$.

(Modify) Running $f \leftarrow v$ on $\mathsf{PK} \setminus \mathsf{W}_q$ yields $\mathsf{PK}_p = \{pk[f := v] \mid pk \in \mathsf{PK} \setminus \mathsf{W}_q\}$, while running it on PK yields $\{pk[f := v] \mid pk \in \mathsf{PK}\}$.

Since $\mathsf{W}_p \cap \mathsf{W}_q = \emptyset$ and $f \in \mathsf{W}_p$, we know that $f \notin \mathsf{W}_q$, so:

$$\begin{aligned}\{pk[f := v] \mid pk \in \mathsf{PK}\} &= \{pk[f := v] \mid pk \in \mathsf{PK} \setminus \mathsf{W}_q\} \times \mathsf{PK} \setminus \{f\} \\ &= \{pk[f := v] \mid pk \in \mathsf{PK} \setminus \mathsf{W}_q\} \times \mathsf{PK} \setminus \{\mathsf{W}_p\} \\ &= \mathsf{PK}_p \times \mathsf{PK} \setminus \{\mathsf{W}_p\}\end{aligned}$$

(Var) Contradictory—we assumed an empty context.

(Par) We know by the first IH that p_1 on $\mathsf{PK} \setminus \mathsf{W}_q$ yields PK_{p_1} and p_1 on PK yields $\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_{p_1}$. Similarly, the second IH tells us that p_2 on $\mathsf{PK}' \setminus \mathsf{W}_q$ yields PK_{p_2} iff p_2 on PK' yields $\mathsf{PK}_{p_2} \times \mathsf{PK}' \setminus \mathsf{W}_{p_2}$.

We must show that $p = p_1 + p_2$ on $\mathsf{PK} \setminus \mathsf{W}_q$ yields PK_p and p on PK yields $\mathsf{PK}_p \times \mathsf{PK} \setminus (\mathsf{W}_{p_1} \cup \mathsf{W}_{p_2})$.

By the strong normalization result (Lemmas 1 and 2), witnessing the existence of a single execution path of a well-typed policy yields the same result as every execution path. We can build such an execution for p by stepping with ParEnter, and then repeatedly stepping with first the left, and then the right congruences according to the executions of p_1 and p_2 resulting from the IH's. Finally, execution terminates by applying ParExit. Hence, we have:

$$\langle p, \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, \mathsf{PK}_{p_1} \cup \mathsf{PK}_{p_2} \rangle$$

and

$$\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, (\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_p) \cup (\mathsf{PK}_{p_2} \times \mathsf{PK} \setminus \mathsf{W}_p) \rangle$$

The result follows from distributing $\times$ over the union using Lemmas 25 and 26.

(Seq) We know by the first IH that p_1 on $\mathsf{PK} \setminus \mathsf{W}_q$ yields PK_{p_1} and p_1 on PK yields $\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_{p_1}$. Similarly, the second IH tells us that p_2 on $\mathsf{PK}' \setminus \mathsf{W}_q$ yields PK_{p_2} and p_2 on PK' yields $\mathsf{PK}_{p_2} \times \mathsf{PK}' \setminus \mathsf{W}_{p_2}$.

We must show that $p = p_1; p_2$ on $\mathsf{PK} \setminus \mathsf{W}_q$ yields PK_p iff p on PK yields $\mathsf{PK}_p \times \mathsf{PK} \setminus (\mathsf{W}_{p_1} \cup \mathsf{W}_{p_2})$.

We can apply the first IH to find that p_1 produces PK_{p_1} on $\mathsf{PK} \setminus \mathsf{W}_q$ iff p_1 on PK produces $\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_{p_1}$.

We must show that p_2 on $\mathsf{PK}_{p_1} \setminus \mathsf{W}_q$ produces PK_{p_2} iff p_2 on PK_{p_1} produces $\mathsf{PK}_{p_2} \times \mathsf{PK}_{p_1} \setminus \mathsf{W}_{p_2}$. This is exactly the second IH, though we must observe that $\mathsf{PK}_{p_1} = \mathsf{PK}_{p_1} \setminus \mathsf{W}_q$.

(Con) We know by the first IH that p_1 on $\mathsf{PK} \setminus \mathsf{W}_{p_2} \setminus \mathsf{W}_q$ yields PK_{p_1} and p_1 on $\mathsf{PK} \setminus \mathsf{W}_{p_2}$ yields $\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_{p_2} \setminus \mathsf{W}_q$. Similarly, the second IH tells us that p_2 on $\mathsf{PK} \setminus \mathsf{W}_{p_1} \setminus \mathsf{W}_q$ yields PK_{p_2} and p_2 on $\mathsf{PK} \setminus \mathsf{W}_{p_1}$ yields $\mathsf{PK}_{p_2} \times \mathsf{PK} \setminus \mathsf{W}_{p_1} \setminus \mathsf{W}_q$.

We must show that $p = p_1 \ {}_{\mathsf{W}_{p_1}}||_{\mathsf{W}_{p_2}} \ p_2$ on $\mathsf{PK} \setminus \mathsf{W}_q$ yields PK_p and p on PK yields $\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_q$.

By the strong normalization result (Lemmas 1 and 2), witnessing the existence of a single execution path of a well-typed policy yields the same result as every execution path. We can build such an execution for p by stepping with ConEnter, and then repeatedly stepping with first the left, and then the right congruences according to the executions of p_1 and p_2 resulting from the IH's. Finally, execution terminates by applying ConExit. Hence, we have:

$$\langle p, \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_{p_1} \times \mathsf{PK}_{p_2}, \mathsf{W}_p \rangle \rangle$$

and

$$\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle \rangle \to^* \langle \mathsf{id}, \langle (\mathsf{PK}_{p_1} \times \mathsf{PK} \setminus \mathsf{W}_{p_2} \setminus \mathsf{W}_p) \times (\mathsf{PK}_{p_2} \times \mathsf{PK} \setminus \mathsf{W}_{p_1} \setminus \mathsf{W}_p), \mathsf{W}_p \rangle \rangle$$

By the associativity and commutativity of times (Lemmas 27 and 26), we have:

$$\mathsf{PK}_{p_1} \times \mathsf{PK}_{p_2} \times \mathsf{PK} \setminus \mathsf{W}_{p_2} \setminus \mathsf{W}_p \times \mathsf{PK} \setminus \mathsf{W}_{p_1} \setminus \mathsf{W}_p$$

After noting that $\mathsf{W}_{p_1} \cap \mathsf{W}_{p_2} = \emptyset$ (because p is well typed), we have:

$$\mathsf{PK}_{p_1} \times \mathsf{PK}_{p_2} \times \mathsf{PK} \setminus (\mathsf{W}_{p_2} \cup \mathsf{W}_{p_2}) \setminus \mathsf{W}_p$$

Of course, as $(\mathsf{W}_{p_2} \cup \mathsf{W}_{p_2}) \subseteq \mathsf{W}_p$ (again due to typing), we have:

$$(\mathsf{PK}_{p_1} \times \mathsf{PK}_{p_2}) \times \mathsf{PK} \setminus \mathsf{W}_p$$

Hence, we can conclude:

$$\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle \rangle \to^* \langle \mathsf{id}, \langle (\mathsf{PK}_{p_1} \times \mathsf{PK}_{p_2}) \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W}_p \rangle \rangle$$

□

Lemma 29 (Permission extension)**.** *If* $\vdash p : (\mathsf{R}, \mathsf{W}_p)$ *and* $\mathsf{W}_p \subseteq \mathsf{W}$ *then*

$$\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W}_p \rangle \rangle$$

iff

$$\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle \rangle.$$

Proof. By induction on the typing derivation.

(Id) Immediate.

(Drop) Immediate, stepping by Drop.

(Match) Immediate, stepping by Match.

(Not) By the IH, wrapping up the derivation with NotEnter and NotInner.

(Modify) Immediate, since $f \in \mathsf{W}_p$ implies $f \in \mathsf{W}$, we can always step by Modify.

(Var) Contradictory—we assumed an empty context.

(Par) By the IHs, wrapping up the derivations with ParEnter and ParExit.

(Seq) By the IH, wrapping up the derivation with SeqEnter and SeqR.

(Con) By the IH—we can change the write permission stored in the con state without affecting the derivation. We wrap up the derivation with ConEnter (with different write permission) and ConExit (restoring the write permission).

□

Lemma 30 (Concurrency serializes)**.** *If* $\vdash p \ {}_{\mathsf{W}_p}||_{\mathsf{W}_q} \ q : (\mathsf{R}, \mathsf{W})$ *and* $\vdash \mathsf{PK} : \tau$ *then* $\langle p \ {}_{\mathsf{W}_p}||_{\mathsf{W}_q} \ q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ *iff* $\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$.

Proof. Because p is well typed, it must be the case that $\mathsf{W} = \mathsf{W}_p \cup \mathsf{W}_q$. We proceed by examining each case separately.

From concurrent to sequential. When going from concurrent composition to sequential composition, we drop ConEnter altogether, and we apply confluence (Lemma 2) to run p first, via ConL. Hence, we have:

$$\langle p, \langle \mathsf{PK} \setminus \mathsf{W}_q, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W}_p \rangle\rangle$$

From packet extension (Lemma 28), we have:

$$\langle p, \langle \mathsf{PK}, \mathsf{W}_p \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W}_p \rangle \rangle$$

From permission extension (Lemma 29) and the fact that $\mathsf{W}_p \subseteq \mathsf{W}$, we have:

$$\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle$$

Now, note that $\mathsf{PK} \setminus \mathsf{W}_p \supseteq (\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p) \setminus \mathsf{W}_p$ by normalization's read-only property (Lemma 1, part (2)). We will apply the previous two lemmas in a similar fashion, but so as to work with PK_p. That is, we know (by monotonicity, Lemma 24) that:

$$\langle q, \langle (\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p) \setminus \mathsf{W}_p, \mathsf{W}_q \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W}_q \rangle \rangle$$

By packet extension (Lemma 28), this means that we have:

$$\langle q, \langle (\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p), \mathsf{W}_q \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK}_q, \mathsf{W}_q \rangle \rangle$$

By permission extension (Lemma 29) and the fact that $\mathsf{W}_q \subseteq \mathsf{W}$, we have:

$$\langle q, \langle (\mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p), \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK}_q, \mathsf{W} \rangle \rangle$$

We can now take our derivation of $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle$ and apply SeqEnter and SeqL to find that $\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \to^* \langle p; q, \langle \mathsf{seq}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle$ and $\langle p; q, \langle \mathsf{seq}\ \langle \mathsf{PK}, \mathsf{W} \rangle \rangle \rangle \to^* \langle \mathsf{id}; q, \langle \mathsf{seq}\ \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle \rangle$. We then step by SeqR to $\langle q, \langle \mathsf{PK}_p \times \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle$. We then know that this steps to $\langle \mathsf{id}, \langle \mathsf{PK}_p \times \mathsf{PK}_q, \mathsf{W} \rangle \rangle$ by our analysis of q—and now we see that this is *exactly* how the concurrent evaluation stepped.

From sequential to concurrent. When going from sequential composition to concurrent composition, we know that $\langle p; q, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle\rangle$; in particular:

$$\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle$$
$$\langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle\rangle$$

We also know that $\vdash p : (\mathsf{R}_p, \mathsf{W}_p)$, so $\mathsf{PK}_p \setminus \mathsf{W}_p \subseteq \mathsf{PK} \setminus \mathsf{W}_p$ by normalization's read-only property (Lemma 1, part (2)). So we can say that $\mathsf{PK}_p = \mathsf{PK}'_p \times \mathsf{PK} \setminus \mathsf{W}_p$ for some PK'_p. This PK'_p models the actual writes done by p: if p drops its packets, it will be the empty set; if p doesn't write at all, it will be the set containing the empty packet (*i.e.* one with no fields). We can now apply packet extension (Lemma 28), showing that $\langle p, \langle \mathsf{PK} \setminus W_q, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_p, \mathsf{W} \rangle\rangle$. We can find this derivation at write permission W_p using permission extension (Lemma 29):

$$\langle p, \langle \mathsf{PK} \setminus W_q, \mathsf{W}_p \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_p, \mathsf{W}_p \rangle\rangle$$

This gives us the derivations to run on p via ConL.

Now we must treat $\langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle\rangle$ to find the steps to run on q via ConR. If PK_p is empty, we can take a shortcut: by normalization (Lemma 1) we can take a shortcut by Lemma 23 to see that our final output PK_q is empty, too. Then by monotonicity (Lemma 24), we know that q will return some set of packets PK''_q when run in the concurrent composition. But when we finally run ConExit, we will multiply $\mathsf{PK}_p \times \mathsf{PK}''_q = \emptyset \times \mathsf{PK}''_q = \emptyset$, and we are done.

Suppose PK'_p isn't empty, and therefore PK_p isn't empty. We have that

$$\langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle\rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}_q, \mathsf{W} \rangle\rangle$$

We know that $\vdash q : (\mathsf{R}_q, \mathsf{W}_q)$, so $\mathsf{PK}_q \setminus \mathsf{W}_q \subseteq \mathsf{PK}_p \setminus \mathsf{W}_q$ by normalization's read-only property (Lemma 1, part (2)). We can therefore say that $\mathsf{PK}_q = \mathsf{PK}'_q \times \mathsf{PK}_p \setminus \mathsf{W}_q$ for

some PK'_q. So we have:

$$\langle q, \langle \mathsf{PK}_p, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_q \times \mathsf{PK}_p \setminus \mathsf{W}_q, \mathsf{W} \rangle \rangle$$

By packet extension (Lemma 28), we have:

$$\langle q, \langle \mathsf{PK}_p \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_q, \mathsf{W} \rangle \rangle$$

Since $\mathsf{PK}'_p \neq \emptyset$, we know that $\mathsf{PK}'_p \setminus \mathsf{W}_p = \{\bot\}$, so $\mathsf{PK}_p \setminus \mathsf{W}_p = \mathsf{PK}'_p \times \mathsf{PK} \setminus \mathsf{W}_p = \mathsf{PK} \setminus \mathsf{W}_p$, yielding:

$$\langle q, \langle \mathsf{PK} \setminus \mathsf{W}_p, \mathsf{W} \rangle \rangle \to^* \langle \mathsf{id}, \langle \mathsf{PK}'_q, \mathsf{W} \rangle \rangle$$

By Lemma 29, we can find this derivation at the write permission W_q, which is enough to give use the ConR derivation.

It remains to see that $\mathsf{PK}'_p \times \mathsf{PK}'_q = \mathsf{PK}_q$. We can expand:

$$\begin{aligned}
\mathsf{PK}_q &= \mathsf{PK}'_q \times \mathsf{PK}_p \setminus \mathsf{W}_q \\
&= \mathsf{PK}'_q \times \mathsf{PK}'_p \times \mathsf{PK} \setminus \mathsf{W}_p \setminus \mathsf{W}_q \\
&= \mathsf{PK}'_q \times \mathsf{PK}'_p \times \mathsf{PK} \setminus \mathsf{W}_p \cup \mathsf{W}_q && \text{because } \mathsf{W}_p \cap \mathsf{W}_q = \emptyset \\
&= \mathsf{PK}'_q \times \mathsf{PK}'_p \times \emptyset && \text{because } \mathsf{W} = \mathsf{W}_p \cup \mathsf{W}_q \\
&= \mathsf{PK}'_q \times \mathsf{PK}'_p
\end{aligned}$$

We form up the final derivation for $p \ {}_{W_p}||_{W_q} \ q$ by beginning with ConEnter, applying the derivations formed above, and then stepping by ConExit, reconstructing that $\mathsf{PK}'_p \times \mathsf{PK}'_q = \mathsf{PK}'$

□

Appendix C

Correctness of the Compilation Algorithms

This section presents the phases of the multipass compilation algorithm described in Section 4, accompanied by proofs of semantic preservation.

C.1 Single-table Compilation

This section presents the full proofs and supporting lemmas of the theorems in Section 4.1.

Definition 22 (NetKAT Sub-languages)**.** *We define the following sub-languages of* $NetKAT$*.*

- $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$*.*
- $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$ *or* p^**.*
- $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$ *or* p^* *or* $\mathsf{sw} = n$*.*
- $p \in NetKAT^{-(\mathsf{sw})}$ *if* p *has the form* $\sum_i \mathsf{sw} = i; p_i$ *and for* $i : 1..n$, $p_i \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$*.*

We call policies $p \in NetKAT^{-\mathsf{dup},\mathsf{sw}\leftarrow}$ *user policies*. This is the set of policies that *NetKAT* programmers may write and that we an implement. We call policies $p \in NetKAT^{sw}$ *switch normal form policies*. Such policies have one component that is specialized to each switch in the network. Compilation is then a series of steps, each refining or eliminating a fragment of the full NetKAT language.

1. **Star elimination.** Transform a *user policy* into an equivalent user policy without Kleene star.

2. **Switch specialization.** Transform a *star-free user policy* into switch normal form.

3. **OpenFlow normal form.** Transform a policy in *switch normal form* into OpenFlow normal form.

Policies in switch normal form are themselves collections of sub-policies, joined by union, each of which is a star-free user policy (*i.e.* does not contain dup, Kleene star, or modifications to the sw field). Note that each of these policy fragments coincides with a table-free, concurrency-free Concurrent NetCore policy.

C.1.1 Star Elimination

Lemma 31 (Star Elimination)**.** *If* $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$*, then there exists* $p' \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ *and* $p \equiv p'$*.*

Proof. The proof begins by showing that p' can be obtained from the normal form used in the completeness theorem. More specifically, let p'' be the policy obtained from p by the normalization construction of Lemma 7 in [2]. By construction, dup can only appear in the normal form of an expression already containing dup, therefore p'' does not contain dup. Moreover, $R(p'') \subseteq I$ and p'' does not contain dup, therefore $R(p'') \subseteq At; P$. Consequently, p'' does not contain Kleene star.

Let us now prove that any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded in the same term by the corresponding test $\mathsf{sw} = sw_i$. Because p does not contain any assignment of the form $\mathsf{sw} \leftarrow sw_i$, it commutes with any test of the form $\mathsf{sw} = sw_i$, and therefore p'' also commutes with any test of the form $\mathsf{sw} = sw_i$. p'' can be written as a sum of $\alpha;p$ for some atoms α and complete assignments p. Suppose for a contradiction that term, α contains a test $\mathsf{sw} = sw_i$, and p contains an assignment $\mathsf{sw} \leftarrow sw_j$, with $sw_i \neq sw_j$. Then

$$\alpha;(\mathsf{sw} = sw_i);p'';(\mathsf{sw} = sw_j) \geq \alpha;p \neq 0$$
$$\alpha;(\mathsf{sw} = sw_j);p'';(\mathsf{sw} = sw_i) = 0$$

but those two terms are also equal, which is a contradiction.

Therefore any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded, in the same term, by the corresponding test $\mathsf{sw} = sw_i$, and can be removed using axiom PA-Filter-Mod to produce the desired p'. Tests and assignments to other fields than sw could appear in between, but we can use the commutativity axioms PA-Mod-Mod-Comm and PA-Mod-Filter-Comm to move the assignment $\mathsf{sw} \leftarrow sw_i$ to just after the test $\mathsf{sw} = sw_i$. □

C.1.2 Switch Normal Form

First, we show that any policy in $NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ can be specialized to a given switch.

Lemma 32 (Switch Specialization)**.** *If $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, then for all switches sw_i, there exists $p' \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ such that $\mathsf{sw} = sw_i;p \equiv \mathsf{sw} = sw_i;p'$.*

Proof. Let g be the unique homomorphism of NetKAT defined on primitive programs by:

$$g(\mathsf{sw} = sw) \triangleq \begin{cases} \mathsf{id} & \text{if } sw = sw_i \\ \mathsf{drop} & \text{otherwise} \end{cases}$$

$$g(f \leftarrow v) \triangleq f \leftarrow v$$

$$g(\mathsf{dup}) \triangleq \mathsf{dup}$$

For every primitive program element x of NetKAT $^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, we have both:

$$\mathsf{sw} = sw_i; x \equiv g(x); \mathsf{sw} = sw_i$$

$$g(x); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(x)$$

Hence, applying Lemma 4.4 in [3] twice shows:

$$\mathsf{sw} = sw_i; p \equiv g(p); \mathsf{sw} = sw_i$$

$$g(p); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(p)$$

By the definition of g, any occurrence of $\mathsf{sw} = v$ in p is replaced by either id or drop in $g(p)$. Moreover, since $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, it follows that $g(p)$ does not contain any occurrence of $\mathsf{sw} = v$ and since $p' = g(p) \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ we also have

$$\mathsf{sw} = sw_i; p \equiv \mathsf{sw} = sw_i; p'$$

□

As there are finitely many switches in a network (codified in Axiom PA-Match-All), we can use Lemma 32 to show that any star-free user policy can be put into switch normal form.

ONF Action Sequence	a	::=	$\mathsf{id} \mid f \leftarrow n; a$
ONF Action Sum	as	::=	$\mathsf{drop} \mid a + as$
ONF Predicate	b	::=	$\mathsf{id} \mid f = n; b$
ONF	ℓ	::=	$as \mid \mathsf{if}\, b\, \mathsf{then}\, as\, \mathsf{else}\, \ell$

Figure 4.2: OpenFlow Normal Form.

Lemma 33 (Switch-Normal-Form)**.** *If* $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ *then there exists* $p' \in NetKAT^{\mathsf{sw}}$ *such that* $p \equiv p'$.

Proof.

	Assertion	**Reasoning**
	p	
$\equiv$	$\mathsf{id}; p$	KA-One-Seq.
$\equiv$	$(\sum_i \mathsf{sw} = i); p$	PA-Match-All.
$\equiv$	$\sum_i (\mathsf{sw} = i; p)$	KA-Seq-Dist-R.
$\equiv$	$\sum_i (\mathsf{sw} = i; p_i)$ where $p_i \in NetKAT^{-\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw}}$	Lemma 32.

□

C.1.3 OpenFlow Normal Form

Finally, we show that any policy in switch normal form is equivalent to a policy in OpenFlow normal form.

Notice that the definition of OpenFlow normal form (ONF) in Figure 4.2 is nearly identical to the definition of switch normal form. The key difference lies in the form of the switch-local policy fragments. In ONF, these policy fragments must be in ONF local form. Hence, we show how to transform a switch-local policy fragment into ONF local form.

The proof proceeds by induction on the structure of the NetKAT policy. We will see that for each combinator, two policy fragments in ONF local form can be joined to produce a policy in ONF local form. After defining several lemmas useful

throughout the remainder of the section, we show that parallel composition, sequential composition, and predicate compilation are all sound. We conclude by showing stating and proving the full compiler soundness theorem.

Coq-based Lemmas. Damien Pous has released a sound and complete decision procedure for Kleene algebras with tests, implemented in the Coq theorem prover [1]. For several lemmas that rely on lengthy program transformations using only the standard KAT axioms, we appeal to this decision procedure. The Coq code for each lemma is reproduced below.

```
Lemma atom_plus_onf_is_onf `{L: laws} n (b : tst n) (p q r : X n n):
  p + [b];q + [!b];r == [b];(p + q) + [!b];(p + r).
Proof. kat. Qed.

Lemma union_is_ite_nf `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [b1];([b2];p1 + [!b2];p2) + [!b1];q ==
  [b1];[b2];p1 + [b1];[!b2];p2 + [!b1];[b2];q + [!b1];[!b2];q.
Proof. kat. Qed.

Lemma demorgans `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [!(b1 \cap b2)];p1 == [!b1];p1 + [!b2];p1.
Proof. kat. Qed.

Lemma ite_nf_is_onf `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [b1];[b2];p1 + [b1];[!b2];p2 + [!b1];[b2];q + [!b1];[!b2];q ==
  [b1 \cap b2];p1 + [!(b1 \cap b2)];([b1];p2 + [!b1];q).
Proof. kat. Qed.

Lemma nested_onf_is_onf `{L: laws}
```

[1] http://perso.ens-lyon.fr/damien.pous/ra

```
  n
  (b1 b2 : tst n)
  (p1 p2 q : X n n):
  [b1];([b2];p1 + [!b2];p2) + [!b1];q ==
  [b1 \cap b2];p1 + [!(b1 \cap b2)];([b1];p2 + [!b1];q).
Proof. kat. Qed.

Lemma union_onf_is_onf `{L: laws}
  n
  (b1 b2 : tst n)
  (a1 a2 p' q' : X n n):
  [b1];a1 + [!b1];p' + [b2];a2 + [!b2];q' ==
  [b1 \cap b2];(a1 + a2) + [!(b1 \cap b2)];([b1];(a1 + q')
  + [!b1];(p' + [b2];a2 + [!b2];q')).
Proof. kat. Qed.

Lemma joined_policies `{L: laws} n (s1 s2 t : X n n):
  ((s1 + s2); t)^* == ((s1; t)^* + (s2; t)^*)^*.
Proof. kat. Qed.
```

Helper lemmas. Several subgoals appear repeatedly in this last part of the proof of compiler correctness. The first simply observes that every test is either true or false (BA-Excluded Middle), and so every program may be broken into a sum, wherein either the test or its negation is sequenced with the program.

Lemma 34 (Predicate Expansion)**.** *For all predicates a, b and policies p, $b;p \equiv a;b;p + \neg a;b;p$.*

Proof.

	Assertion	Reasoning
1	$b; p$	
2	$\equiv \mathsf{id}; b; p$	KA-One-Seq
3	$\equiv (a + \neg a); b; p$	BA-Excluded-Middle
Goal	$\equiv a; b; p + \neg a; b; p$	KA-Seq-Dist-R. $\square$

Consider ONF as a series of nested if statements for a moment. During the proofs, we will often conclude that the body of *both* branches of the topmost if statement are themselves nested if statements in ONF. In order to show that the expression as a whole is indeed in ONF, we must show that the structure is equivalent to a single series of nested if statements.

Lemma 35 (Nested ONF is ONF)**.** *For all ONF predicates a and policies p, q in ONF, there exists a policy r in ONF such that $r \equiv a; p + \neg a; q$.*

Proof. By induction on the structure of p.

Case 1. We have $p = as_1$, and the result is immediate.

Case 2. We have $p = b; as + \neg b; q$.

Assertion	Reasoning
$a; (b; as + \neg b; q) + \neg a; p'$ $\equiv (a; b); as + \neg(a; b); (a; q + \neg a; p')$	Coq Lemma nested_onf_is_onf.

Applying the induction hypothesis to $(a; q + \neg a; p')$ yields r' in ONF. Hence, our goal is satisfied with $(a; b); as + \neg(a; b); r'$. $\square$

Finally, there are several commutativity conditions that are essential to reasoning about NetKAT program transformations.

Lemma 36 (Negative Field Commutativity)**.** *For all fields f_1, f_2 and values n_1, n_2 such that $f_1 \neq f_2$, $f_1 \leftarrow n_1; \neg f_2 = n_2 \equiv \neg f_2 = n_2; f_1 \leftarrow n_1$.*

Proof. By PA-Mod-Mod-Comm, we have $f_1 \leftarrow n_1; f_2 = n_2 \equiv f_2 = n_2; f_1 \leftarrow n_1$. The desired result then follows from [26, Lemma 2.3.1]. □

Lemma 37 (Action Atom Seq Filter Commutativity)**.** *For all ONF action sequences a_1, fields f, and values n, one of the following holds:*

- $a_1; f = n \equiv f = n; a_1$ *and* $a_1; \neg f = n \equiv \neg f = n; a_1$,
- $a_1; f = n \equiv a_1$ *and* $a_1; \neg f = n \equiv \mathsf{drop}$, *or*
- $a_1; f = n \equiv \mathsf{drop}$ *and* $a_1; \neg f = n \equiv a_1$.

Proof. By induction on the structure of a_1. We have two cases. The base case, $a_1 \equiv \mathsf{id}$, is trivial. We continue with the induction case, $a_1 \equiv f_1 \leftarrow n_1; a_1'$. Applying the induction hypothesis to a_1', f, and n yields three new cases:

Case 1. We have $f_1 \leftarrow n_1; a_1'; f = n \equiv f_1 \leftarrow n_1; f = n; a_1'$ and $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv f_1 \leftarrow n_1; \neg f = n; a_1'$. There are three cases:

Case 1a. If $f_1 \neq f$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv f = n; f_1 \leftarrow n_1; a_1'$ by PA-Mod-Filter-Comm. By Lemma 36, we also have $f_1 \leftarrow n_1; \neg f_2 = n_2 \equiv \neg f_2 = n_2; f_1 \leftarrow n_1$.

Case 1b. If $f_1 \equiv f$ and $n \equiv n_1$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv f_1 \leftarrow n_1; a_1'$ by PA-Mod-Filter. We also have:

	Assertion	**Reasoning**
1	$f_1 \leftarrow n_1; \neg f = n; a_1'$	
2	$\equiv f_1 \leftarrow n_1; f = n; \neg f = n; a_1'$	PA-Mod-Filter.
3	$\equiv f_1 \leftarrow n_1; \mathsf{drop}; a_1'$	BA-Contra.
Goal	$\equiv \mathsf{drop}$	KA-Zero-Seq and KA-Seq-Zero.

Case 1c. If $f_1 \equiv f$ and $n \neq n_1$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv \mathsf{drop}$ by PA-Mod-Filter and PA-Contra. We also have:

	Assertion	Reasoning
1	$f \leftarrow n_1; \neg f = n; a_1'$	
2	$\equiv \mathsf{drop} + f \leftarrow n_1; \neg f = n; a_1'$	KA-Plus-Zero, KA-Plus-Comm.
3	$\equiv f \leftarrow n_1; \mathsf{drop}; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	KA-Seq-Zero, KA-Zero-Seq.
4	$\equiv f \leftarrow n_1; f = n_1; f = n; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	PA-Contra.
5	$\equiv f \leftarrow n_1; f = n; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	PA-Mod-Filter-Comm.
6	$\equiv f \leftarrow n_1; (f = n + \neg f = n); a_1'$	KA-Seq-Dist-L, KA-One-Seq, KA-Seq-Dist-R.
Goal	$\equiv f \leftarrow n_1; a_1'$	BA-Excluded-Middle, KA-Seq-One.

Case 2. We know that

- $a_1'; f = n \equiv a_1'$, and
- $a_1'; \neg f = n \equiv \mathsf{drop}$.

Hence, by substitution and KA-Seq-Zero,

- $f_1 \leftarrow n_1; a_1'; f = n \equiv f_1 \leftarrow n_1; a_1'$, and
- $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv \mathsf{drop}$.

Case 3. We know that

- $a_1'; f = n \equiv \mathsf{drop}$, and
- $a_1'; \neg f = n \equiv a_1'$.

Hence, by substitution and KA-Seq-Zero,

- $f_1 \leftarrow n_1; a_1'; f = n \equiv \mathsf{drop}$, and
- $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv f_1 \leftarrow n_1; a_1'$. □

Lemma 38 (Action Atom Seq Predicate Commutativity)**.** *For all ONF action sequences a_1 and ONF predicates b_1, one of three cases holds:*

- *there exists an ONF predicate* b_2 *such that* $a_1; b_1 \equiv b_2; a_1$ *and* $a_1; \neg b_1 \equiv \neg b_2; a_1$,
- $a_1; b_1 \equiv a_1$ *and* $a_1; \neg b_1 \equiv \mathsf{drop}$, *or*
- $a_1; b_1 \equiv \mathsf{drop}$ *and* $a_1; \neg b_1 \equiv a_1$.

Proof. By induction on the structure of b_1. The base case, $b_1 \equiv \mathsf{id}$, is trivial, leaving the inductive case $b_1 \equiv f = n; b_1'$. We have $a_1; f = n; b_1'$. Applying Lemma 37 to $a_1; f = n$ yields three cases.

Case 1. We have that $a_1; f = n \equiv f = n; a_1$. Hence, $a_1; f = n; b_1' \equiv f = n; a_1; b_1'$ and the goal follows from an application of the induction hypothesis.

The same reasoning resolves the negative case, where $a_1; \neg f = n \equiv \neg f = n; a_1$.

Case 2. We have that $a_1; f = n \equiv a_1$. Hence, $a_1; f = n; b_1' \equiv a_1; b_1'$ and the goal follows from an application of the induction hypothesis.

The negative case, where $a_1; \neg f = n \equiv \mathsf{drop}$, follows from rewriting and KA-Seq-Zero.

Case 3. We have that $a_1; f = n \equiv \mathsf{drop}$. Hence, $a_1; f = n; b_1' \equiv \mathsf{drop}$ by KA-Seq-Zero.

The negative case, where $a_1; \neg f = n \equiv a_1$, proceeds as follows. By rewriting, we have $a_1; \neg f = n; b_1' \equiv a_1; b_1'$ and the goal follows from an application of the induction hypothesis. □

Parallel Composition Preserves ONF. In this section, we show that the parallel composition of two policies in ONF is itself equivalent to a policy in ONF. The proof proceeds by mutual induction on the structure of both subpolicies, and the section presents supporting lemmas leading up to the final result.

Lemma 39 (Action Plus Action is ONF)**.** *For all ONF action sums* as_1 *and* as_2, *there exists an ONF action sum* as_3 *such that* $as_1 + as_2 \equiv as_3$.

Proof. Both as_1 and as_2 can take one of two forms, drop or $a + as'$, leading to four cases. The goal follows immediately from KA-Plus-Zero in all but the final case.

Case 4. We have $as_1 \equiv a_1 + as'_1$ and $as_2 \equiv a_2 + as'_2$. $as_3 \equiv a_1 + a_2 + as'_1 + as'_2$, which can be put into normal form via list concatenation. □

Lemma 40 (Action Plus ONF is ONF)**.** *For all ONF action sums as_1 and policies $p \equiv b; as_2 + \neg b; q$ in ONF, there exists a policy r in ONF such that $r \equiv as_1 + p$.*

Proof. By induction on p.

Case 1. We have $p \equiv as_2$. The goal follows immediately from Lemma 39.

Case 2. We have $p \equiv b; as_2 + \neg b; q$.

	Assertion	**Reasoning**
1	$as_1 + b; as_2 + \neg b; q$	
2	$\equiv b; as_1 + \neg b; as_1 + b; as_2 + \neg b; q$	Lemma 34.
3	$\equiv (b; as_1) + (b; as_2) + (\neg b; as_1) + (\neg b; q)$	KA-Plus-Comm.
4	$\equiv b; (as_1 + as_2) + \neg b; (as_1 + q)$	KA-Seq-Dist-L.

The goal then follows from an application of the induction hypothesis to $(as_1 + q)$. □

Lemma 41 (Parallel Composition Preserves ONF)**.** *For all policies p, q in OpenFlow normal form, there exists r such that $p + q \equiv r$ and r is in OpenFlow normal form.*

Proof. By induction on p and split in to cases based on the structure of q.

Case 1. We have $p \equiv as_1$ and $q \equiv as_2$, and $r \equiv as_1 + as_2$. The goal follows from Lemma 39.

Case 2. We have $p \equiv as_1$ and $q \equiv b; as_2 + \neg b; q'$, and $r \equiv as_1 + b; as_2 + \neg b; q'$. The goal follows from Lemma 40.

Case 3. We have $p \equiv b; as_1 + \neg b; p'$ and $q \equiv as_2$, and $r \equiv b; as_1 + \neg b; p' + as_2$. The goal follows from Lemma 40 and KA-Plus-Comm.

Case 4. We have $p \equiv b_1; as_1 + \neg b_1; p'$ and $q \equiv b_2; as_2 + \neg b_2; q'$, and $r \equiv b_1; as_1 + \neg b_1; p' + b_2; as_2 + \neg b_2; q'$.

	Assertion	**Reasoning**
1	$b_1; as_1 + \neg b_1; p' + b_2; as_2 + \neg b_2; q'$	
2	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; (as_1 + q') + \neg b_1; (p' + b_2; as_2 + \neg b_2; q'))$	Coq Lemma union_onf_is_onf.
3	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; (p' + b_2; as_2 + \neg b_2; q'))$	(for some r_1 in ONF) by Lemma 40.
4	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; (p' + q))$	Substitute q.
5	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; r_2)$	IH.
6	$(b_1; b_2); (as_1 + as_2) + \neg(b_1; b_2); r_3$	Lemma 35. □

Sequential Composition Preserves ONF. In this section, we show that the sequential composition of two policies in ONF is itself equivalent to a policy in ONF. The proof proceeds by mutual induction on the structure of both subpolicies and the structure of ONF action sums in the left-hand policy. The section presents supporting lemmas leading up to the final result.

Lemma 42 (Conjunct Seq Conjunct is ONF)**.** *For all ONF action sequences a_1, a_2, there exists an ONF action sequence $a_3 \equiv a_1; a_2$.*

Proof. By induction on the structure of a_1.

Case 1. We have $a_1 \equiv \mathsf{id}$ and $a_3 \equiv \mathsf{id}; a_2 \equiv a_2$ by KA-One-Seq.

Case 2. We have $a_1 \equiv f \leftarrow n; a_1'$ and $a_3 \equiv (f \leftarrow n; a_1'); a_2$.

	Assertion	Reasoning
1	$a_3 \equiv (f \leftarrow n; a_1'); a_2$	
2	$\equiv f \leftarrow n; a_1'; a_2$	KA-Seq-Assoc
Goal	$\equiv f \leftarrow n; a_3'$	IH $\square$

Lemma 43 (Conjunct Seq Action is ONF)**.** *For all ONF action sequences a and ONF action sums as_1, there exists an ONF action sum $as_2 \equiv a; as_1$.*

Proof. We begin with induction on as_1. The first case is trivial: $as_2 \equiv a; \mathsf{drop} \equiv \mathsf{drop}$ by KA-Seq-Zero. In the second case, $as_1 \equiv a_1 + as_1'$.

	Assertion	Reasoning
1	$as_2 \equiv a; (a_1 + as_1')$	
2	$\equiv a; a_1 + a; as_1'$	KA-Seq-Dist-L.
3	$\equiv a_2 + a; as_1'$	Lemma 42.
Goal	$\equiv a_2 + as_2'$	IH $\square$

Lemma 44 (Action Seq Action is ONF)**.** *For all ONF action sums as_1 and as_2, there exists an ONF action sum as_3 such that $as_1; as_2 \equiv as_3$.*

Proof. By induction on as_1. In the first case, $as_1 \equiv \mathsf{drop}$, and the goal follows from KA-Zero-Seq. In the latter case, $as_1 \equiv a_1 + as_1'$.

	Assertion	Reasoning
1	$as_3 \equiv (a_1 + as_1'); as_2$	
2	$\equiv a_1; as_2 + as_1'; as_2$	KA-Seq-Dist-R.
3	$\equiv as_{31} + as_1'; as_2$	Lemma 43.
4	$\equiv as_{31} + as_{32}$	IH.
Goal	$\equiv as_3'$	Lemma 39. $\square$

Lemma 45 (Action Atom Seq ONF is ONF)**.** *For all ONF action sequences a_1 and policies p in ONF, there exists a policy q in ONF such that $q \equiv a_1; p$.*

Proof. By induction on p. The base case, $p \equiv as$, is discharged by Lemma 43. That leaves the inductive case, $p \equiv b; as + \neg b; p'$ and $q \equiv a_1; (b; as + \neg b; p')$.

	Assertion	Reasoning
1	$q \equiv a_1; (b; as + \neg b; p')$	
2	$\equiv a_1; b; as + a_1; \neg b; p'$	KA-Seq-Dist-L.

Applying Lemma 38 to a_1 and b yields three cases:

Case 1. We have that $a_1; b \equiv b; a_1$ and $a_1; \neg b \equiv \neg b; a_1$. After rewriting line 2 from the proof table above, we have:

	Assertion	Reasoning
1	$b; a_1; as + \neg b; a_1; p'$	
2	$b; a_1; as + \neg b; q'$	(for some q' in ONF) by IH
2	$b; as_2 + \neg b; q'$	(for some as_2) by Lemma 43

Case 2. We have that $a_1; b \equiv a_1$ and $a_1; \neg b \equiv \mathsf{drop}$. After rewriting, we have:

	Assertion	Reasoning
1	$a_1; as + \mathsf{drop}; p'$	
2	$a_1; as$	KA-Zero-Seq and KA-Plus-Zero.
3	as_2	(for some as_2) by Lemma 43

Case 3. We have that $a_1; b \equiv \mathsf{drop}$ and $a_1; \neg b \equiv a_1$. After rewriting, we have:

	Assertion	Reasoning
1	$\mathsf{drop}; as + a_1; p'$	
2	$a_1; p'$	KA-Zero-Seq and KA-Plus-Zero.
3	q'	by IH □

Lemma 46 (Action Seq ONF is ONF)**.** *For all ONF action sums as and policies p in ONF, there exists a policy q in ONF such that $q \equiv as; p$.*

Proof. By induction on the structure of as. The base case, $as = \mathsf{drop}$, is trivial. This leaves the inductive case, where $as = a + as'$.

	Assertion	Reasoning
1	$(a + as'); p$	
2	$\equiv a; p + as'; p$	KA-Seq-Dist-R.
3	$\equiv r + as'; p$	Lemma 45.
4	$\equiv r + q'$	IH.

The goal then follows from applying Lemma 41. □

Lemma 47 (Sequential Composition Preserves ONF)**.** *For all policies p, q in OpenFlow normal form, there exists r such that $p; q \equiv r$ and r is in OpenFlow normal form.*

Proof. By induction on p.

Case 1. We have $p = as_1$ and $r \equiv as_1; q$. The result follows from Lemma 46.

Case 2. We have $p = b; as_1 + \neg b; p'$ and $r \equiv (b; as_1 + \neg b; p'); q$.

	Assertion	Reasoning
1	$(b; as_1 + \neg b; p'); q$	
2	$\equiv b; as_1; q + \neg b; p'; q$	KA-Seq-Dist-R.
3	$\equiv b; r_1 + \neg b; p'; q$	(for some r_1 in ONF) by Lemma 46.
4	$\equiv b; r_1 + \neg b; r_2$	IH.

The result then follows from applying Lemma 35. □

Predicate Compilation is Sound. We can also see that NetKAT predicates have equivalents in ONF.

Lemma 48 (Filter is ONF)**.** *For all policies $p = a$, there exists a policy q in ONF such that $p \equiv q$.*

Proof. By induction on the structure of a. The cases of id and drop are immediate.

Case: $f = n$.

	Assertion	Reasoning
1	$f = n$	
2	$\equiv f = n + \mathsf{drop}$	KA-Plus-Zero.
Goal	$\equiv f = n; \mathsf{id} + +\neg f = n; \mathsf{drop}$	KA-Seq-Zero, KA-Seq-One.

Case: $\neg b$.

	Assertion	Reasoning
1	$\neg b$	
2	$\equiv \neg b; \mathsf{id}$	KA-Seq-One.
3	$\equiv \mathsf{drop} + \neg b; \mathsf{id}$	KA-Plus-Zero, KA-Plus-Comm.
Goal	$\equiv b; \mathsf{drop} + \neg b; \mathsf{id}$	KA-Seq-Zero.

Case: $b + c$. From the induction hypothesis, we have $q \equiv q' + q''$. The result then follows from an application of Lemma 41.

Case: $b; c$. From the induction hypothesis, we have $q \equiv q'; q''$. The result then follows from an application of Lemma 47. □

Compiler Soundness. Finally, we show that any switch fragment of a policy in switch normal form has an equivalent policy in OpenFlow normal form. Hence, any user policy has an equivalent policy in OpenFlow normal form.

Lemma 49. *Switch-local Compilation*

If $p \in NetKAT^{-(\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw})}$ *then there exists a policy* p' *such that* $p \equiv p'$ *and* $p' \in \mathit{ONF}$.

Proof. By induction on the structure of p.

Case 1. We have $p = a$. The goal follows from applying Lemma 48.

Case 2. We have $p = f \leftarrow n$ (and $f \neq \mathsf{sw}$). Applying KA-Seq-One yields $p \equiv f \leftarrow n; \mathsf{id}$, which satisfies our goal.

Case 3. We have $p = q + r$. Applying the induction hypothesis to q and r yields $p \equiv q' + r'$, after substitution. The goal then follows from Lemma 41.

Case 4. We have $p = q; r$. Applying the induction hypothesis to q and r yields $p \equiv q'; r'$, after substitution. The goal then follows from Lemma 47.

Case 5. We have $p = p'^*$. This case is inconsistent with the hypothesis ($p'^* \notin NetKAT^{-\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw}}$). □

Theorem 8 (User policies can be compiled to ONF)**.** *For all user policies $p \in NetKAT^{-\mathsf{dup},\mathsf{sw}\leftarrow}$ there exists a policy $p' \in \mathit{ONF}$ such that $p \equiv p'$.*

Proof. By Lemmas Star-Elimination, Switch-Normal-Form and Switch-Local-Compilation. □

C.1.4 Optimizations

Lemma 50 (If Compress)**.** $\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \equiv \mathsf{if}\ b_1 + b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$

Proof. By desugaring the *if* statements and then applying boolean algebra and distributivity axioms. □

Lemma 51. *Fall-through Elimination*

If $b_1 \leq b_2$ then $\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \equiv \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$ *.*

Proof.

$$
\begin{aligned}
& \mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \\
\equiv\ & \mathsf{if}\ b_1 + b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell && \text{by If Compress} \\
\equiv\ & \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell && \text{by } b_1 \leq b_2 \quad \square
\end{aligned}
$$

C.1.5 Compiling to Physical Tables

Lemma 52. *For all* crossbar *policies* c*, if* $c = \mathsf{if}\ \mathsf{action} = v\ \mathsf{then}\ p\ \mathsf{else}\ c'$*, then for all* $v' \neq v$*,* $\neg\mathsf{action} = v'; c' \equiv c' \equiv c'; \neg\mathsf{do}_v = 1$*.*

Proof. Induction on c', relying on Lemma 56 and the fact that v does not appear in a predicate in c' by construction of c'.

□

Lemma 53. *For all* action *policies* t*, if* $t = \mathsf{if}\ \mathsf{action} = v\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{id} \mid\mid t'$*, then for all* $v' \neq v$*,* $\neg\mathsf{action} = v'; t' \equiv t'$*. Similarly, if* $v = v'$*, then* $\mathsf{action} = v'; t' \equiv \mathsf{id}$*.*

Proof. Induction on t', relying on Lemma 56 and the fact that v does not appear in a predicate in t' by construction of t'.

□

Lemma 54 (Single-table compilation from ONF to physical tables is semantics preserving)**.** *For all ONF tables ons without packet duplication, let* $m, c,$*, and* t *be match, crossbar, and action stages such that* $m; c; t = crossbar(ons)$*, and let fs be the fresh metadata fields introduced by crossbar, with* $z = \Pi_{f \in fs} f \leftarrow 0$ *zeroing these fields. The following equivalence holds.*

$$z; m; c; t; \equiv; z; ons; z$$

Proof. Induction on *ons*. The base case is immediate.

Case $ons = \mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ ons'$**.** From the application of *crossbar*, we have that

- $m'; c'; t' = crossbar(ons')$
- $m; c; t = \mathsf{if}\ a\ \mathsf{then}\ action \leftarrow map(p)\ \mathsf{else}\ m; \mathsf{if}\ action = map(p)\ \mathsf{then}\ \mathsf{do}_{map(p)} \leftarrow 1\ \mathsf{else}\ c; \mathsf{if}\ \mathsf{do}_{map(p)} = 1\ \mathsf{then}\ p\ \mathsf{else}\ t$

From the IH, we have that $m'; c'; t' \equiv ons'$.

	Assertion	Reasoning
	$m; c; t$	
$\equiv$	if a then action $\leftarrow v$ else m;	
	if action $= v$ then $\mathsf{do}_v \leftarrow 1$ else c;	
	if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Substitution]
$\equiv$	$(a;$ action $\leftarrow v + \neg a; m);$	
	$(\text{action} = v; \mathsf{do}_v \leftarrow 1 + \neg\text{action} = v; c);$	
	if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Desugar]
$\equiv$	$(a;$ action $\leftarrow v;$ action $= v; \mathsf{do}_v \leftarrow 1$	
$+$	$a;$ action $\leftarrow v; \neg$action $= v; c$	
$+$	$\neg a; m;$ action $= v; \mathsf{do}_v \leftarrow 1$	
$+$	$\neg a; m; \neg$action $= v; c);$	
	if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Dist-L, Dist-R]
$\equiv$	$(a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1$	
$+$	drop	
$+$	$\neg a; m;$ action $= v; \mathsf{do}_v \leftarrow 1$	
$+$	$\neg a; m; \neg$action $= v; c);$	
	if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Mod-Match, BA-Contra]
$\equiv$	$(a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1$	
$+$	drop	
$+$	drop	
$+$	$\neg a; m; \neg$action $= v; c);$	
	if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[v is unique, Mod-Match]
$\equiv$	$a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1;$ if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	
$+$	$\neg a; m; \neg$action $= v; c;$ if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Dist-R]
$\equiv$	$a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p$	
$+$	$\neg a; m; \neg$action $= v; c;$ if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Mod-Match, Lemmas 56, 53]
$\equiv$	$a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p$	
$+$	$\neg a; m; c; \neg\mathsf{do}_v = 1;$ if $\mathsf{do}_v = 1$ then p else id $\|\|\, t$	[Lemma 52]
$\equiv$	$a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p$	
$+$	$\neg a; m; c; \neg\mathsf{do}_v = 1; t$	[BA-Contra]
$\equiv$	$a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p$	
$+$	$\neg a; m; c; t$	[Lemma 53]

Substituting equals for equals, it then follows that

$$z; m; c; t; z \equiv z; (a; \text{action} \leftarrow v; \mathsf{do}_v \leftarrow 1; p + \neg a; m; c; t); z$$

	Assertion	Reasoning
$\equiv$	$z; (a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p + \neg a; m; c; t); z$	
$\equiv$	$z; a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; p; z + z; \neg a; m; c; t; z$	[Dist-L, Dist-R]
$\equiv$	$z; a;$ action $\leftarrow v; \mathsf{do}_v \leftarrow 1; z; p + z; \neg a; m; c; t; z$	[Freshness]

$$
\begin{array}{lll}
\equiv & z; a; \mathsf{action} \leftarrow v; z; p + z; \neg a; m; c; t; z & \text{[Mod-Mod]} \\
\equiv & z; a; z; p + z; \neg a; m; c; t; z & \text{[Mod-Mod]} \\
\equiv & z; a; p; z + z; \neg a; m; c; t; z & \text{[Freshness]} \\
\equiv & z; a; p; z + \neg a; z; m; c; t; z & \text{[Freshness]} \\
\equiv & z; a; p; z + \neg a; z; ons'; z & \text{[Substitution (IH)]} \\
\equiv & z; a; p; z + z; \neg a; ons'; z & \text{[Freshness]} \\
\equiv & z; (a; p + \neg a; ons'); z & \text{[Dist-L, Dist-R]} \\
\equiv & z; (\mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ ons'); z & \text{[Sugar]}
\end{array}
$$

□

C.2 Pipeline Compilation

This section presents the phases of the pipeline compilation algorithms described in Section 4.2, accompanied by proofs of semantic preservation. There are two minor departures from that presentation:

- In this presentation of multicast refactoring, we replace s with an annotated type $\tau \in \mathcal{P}(\mathsf{Field}) \times \mathcal{P}(\mathsf{Field}) \times \mathsf{Int}$, where the final integer field represents the number of multicast occurrences in future updates to a table variable.
- In this presentation of table fitting, we use an additional annotation on tables to describe the expected number of rules, rather than supplying an auxiliary map.

C.2.1 Useful Lemmas

Lemma 55. *For all predicates a and b, $\neg a; b \equiv \neg a + \neg b$.*

Proof.

```
(1) First we show that ab + (-a + -b) == 1.


    ab + -a + -b

    [Seq-1, 1-Seq]
```

```
ab + -a1 + 1-b

[BA-Excl-Mid]

ab + -a(b + -b) + (a + -a)-b

[Seq-Dist-R]

ab + -a(b + -b) + a-b + -a-b

[Seq-Dist-L]

ab + -ab  + -a-b + a-b + -a-b

[Plus-Comm, Plus-Idem]

ab + -ab + a-b + -a-b

[Seq-Dist-L, Seq-Dist-R]

(a + -a)(b + -b)

[BA-Excl-Mid]

11

[BA-Seq-Idem]

1
```

(2) Next, we show that ab(-a + -b) == 0.

```
ab(-a + -b)

[Seq-Dist-L]

ab-a + ab-b

[BA-Contra]

ab-a + a0

[Seq-Zero, Plus-Zero]

ab-a

[BA-Seq-Comm]

a-ab
```

```
    [BA-Contra]
    0b
    [Zero-Seq]
    0
```

Now, we have:

```
    -(ab)
    [Seq-One]
    -(ab)1
    [Substitution (1)]
    -(ab)(ab + -a + -b)
    [Seq-Dist-L]
    -(ab)ab + -(ab)-a + -(ab)-b
    [BA-Contra]
    0 + -(ab)-a + -(ab)-b
    [Seq-Dist-L]
    0 + -(ab)(-a + -b)
    [Substitution (2)]
    ab(-a + -b) + -(ab)(-a + -b)
    [Seq-Dist-R]
    (ab + -(ab))(-a + -b)
    [BA-Contra]
    -a + -b
```

□

Lemma 56. *For all fields f and values v, v', if $v \neq v'$ then $f = v; \neg f = v' \equiv f = v$.*

Proof.

```
f = v; -(f = v')

f = v; -(f = v') + f = v; -(f = v)

f = v; (-(f = v') + -(f = v))

f = v; -(f = v'; f = v)

f = v; -(drop)

f = v; -(drop); id

f = v; (-(drop) + drop)

f = v; id

f = v
```

□

C.2.2 Refactoring Parallel Composition

Current and proposed switching architectures support a limited form of multicast—packets may be duplicated at only a limited number of stages. This section describes a compilation that restructures parallel composition to target the RMT pipeline. There are three stages:

- **Analyze.** The first step is to analyze the given policy to determine the number of metadata fields required to compile it.
- **Compile multicast.** The second step refactors the policy to consolidate packet duplication to a single stage.
- **Fit to tables.** The final step fits a refactored policy to a set of tables in hardware.

Definition 23.

Let r range over concurrent NetCore terms excluding parallel composition.

predicate sequences	t	::=	$f = 1 \mid t; f = 1$
assignment sequences	s	::=	$\mathsf{id} \mid s; f \leftarrow 1$
assignment sums	m	::=	$\mathsf{drop} \mid m + x : \tau \mid m + s$
if statement sequences	n	::=	$x : \tau \mid n; r \mid n; \mathsf{if}\ t\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$
zeroing assignment sequences	z	::=	$\mathsf{id} \mid z; f \leftarrow 0$

For the sake of presentation, we repeat several pieces of compilation machinery here.

Table bindings	b	$\in$	$\mathsf{Table} \to$ R-Fragment Policy
Closing substitutions	T	$\in$	$\mathsf{B} \to \mathsf{Policy} \to \mathsf{Policy}$
Binding transformers	θ	$\in$	$\mathsf{B} \to \mathsf{B}$

Table bindings b represent updates to these tables—an SDN controller can be thought of as emitting a sequence of table bindings. One might think of the first table binding as *initializing* the tables in a policy, and subsequent bindings as installing new rules to each table. We write $T_b\ p$ to replace tables in the policy p with corresponding values drawn from b.

Lemma 57 (Qualification produces well-formed egress stages)**.** *For all egress stages n_1 and n_2, predicates a, and table bindings b, if*

- $n_2 = \mathsf{qualify}\ a\ n_1$, *and*
- a *commutes with all atomic operations in* $T_b\ n_1$,

then

$$T_b\ (\mathsf{if}\ a\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id}) \equiv T_b\ n_2.$$

Proof. The proof proceeds by induction on n_1.

Case $n_1 = \mathsf{id}$**.** Immediate from the definitions of qualify and if statements.

Case $n_1 = x : \tau$**.** We immediately have that $n_2 = \mathsf{if}\ a\ \mathsf{then}\ (x : \tau)\ \mathsf{else}\ \mathsf{id}$, from the definition of qualify.

Case $n_1 = n; r$**.** We have that $n_2 = (\mathsf{qualify}\ a\ n); \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$.

	Assertion	**Reasoning**
	$T_b\ (\mathsf{qualify}\ a\ n); \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	
$\equiv$	$T_b\ (\mathsf{qualify}\ a\ n); T_b\ \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	Definition of T .
$\equiv$	$T_b\ \mathsf{if}\ a\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{id}; T_b\ \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	IH.
$\equiv$	$T_b\ \mathsf{if}\ a\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{id}; \mathsf{if}\ a\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	Definition of T .
$\equiv$	$T_b\ (a; n + \neg a; \mathsf{id}); (a; r + \neg a; \mathsf{id})$	Desugar if statements.
$\equiv$	$T_b\ a; n; a; r + \neg a; \mathsf{id}; a; r + a; n; \neg a; \mathsf{id} + \neg a; \mathsf{id}; \neg a; \mathsf{id}$	Dist-L, Dist-R.
$\equiv$	$T_b\ a; a; n; r + \neg a; \mathsf{id}; a; r + a; \neg a; n; \mathsf{id} + \neg a; \mathsf{id}; \neg a; \mathsf{id}$	KAT-Commute.
$\equiv$	$T_b\ a; n; r + \neg a; \mathsf{id}; a; r + a; \neg a; n; \mathsf{id} + \neg a; \mathsf{id}$	BA-Seq-Idem.
$\equiv$	$T_b\ a; n; r + \mathsf{drop} + \mathsf{drop} + \neg a; \mathsf{id}$	BA-Contra.
$\equiv$	$T_b\ a; n; r + \neg a; \mathsf{id}$	KA-Plus-Zero.
$\equiv$	$T_b\ \mathsf{if}\ a\ \mathsf{then}\ n; r\ \mathsf{else}\ \mathsf{id}$	Sugar if statements.

Case $n_1 = n; \mathsf{if}\ c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$**.** We have that $n_2 = (\mathsf{qualify}\ a\ n); \mathsf{if}\ a; c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$.

	Assertion	**Reasoning**
	$T_b\ (\mathsf{qualify}\ a\ n); \mathsf{if}\ a; c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	
$\equiv$	$T_b\ (\mathsf{qualify}\ a\ n); T_b\ \mathsf{if}\ a; c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	Definition of T .
$\equiv$	$T_b\ \mathsf{if}\ a\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{id}; T_b\ \mathsf{if}\ a; c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	IH.
$\equiv$	$T_b\ \mathsf{if}\ a\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{id}; \mathsf{if}\ a; c\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}$	Definition of T .
$\equiv$	$T_b\ (a; n + \neg a; \mathsf{id}); (a; c; r + \neg(a; c); \mathsf{id})$	Desugar if statements.
$\equiv$	$T_b\ a; n; a; c; r + \neg a; \mathsf{id}; a; c; r$	
	$+ a; n; \neg(a; c); \mathsf{id} + \neg a; \mathsf{id}; \neg(a; c); \mathsf{id}$	Dist-L, Dist-R.
$\equiv$	$T_b\ a; n; a; c; r + a; n; \neg(a; c); \mathsf{id} + \neg a; \mathsf{id}; \neg(a; c); \mathsf{id}$	BA-Contra, KA-Seq-One, KA-Plus-Zero.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg(a; c); \mathsf{id} + \neg a; \mathsf{id}; \neg(a; c); \mathsf{id}$	KAT-Commute, BA-Seq-Idem.
$\equiv$	$T_b\ a; n; c; r + a; n; (\neg a + \neg c); \mathsf{id} + \neg a; \mathsf{id}; (\neg a + \neg c); \mathsf{id}$	De Morgan's law.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg a; \mathsf{id} + a; n; \neg c; \mathsf{id}$	
	$+ \neg a; \mathsf{id}; \neg a; \mathsf{id} + \neg a; \mathsf{id}; \neg c; \mathsf{id}$	Dist-L, Dist-R.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; \mathsf{id}; \neg a; \mathsf{id} + \neg a; \mathsf{id}; \neg c; \mathsf{id}$	KAT-Commute, BA-Contra, KA-Plus-Zero.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; \mathsf{id} + \neg a; \mathsf{id}; \neg c; \mathsf{id}$	BA-Seq-Idem.

$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; (c + \neg c); \mathsf{id} + \neg a; \mathsf{id}; \neg c; \mathsf{id}$	KA-Seq-One, BE-Excl-Mid.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; c; \mathsf{id} + \neg a; \neg c; \mathsf{id} + \neg a; \mathsf{id}; \neg c; \mathsf{id}$	Dist-L, Dist-R.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; c; \mathsf{id} + \neg a; \neg c; \mathsf{id}$	KA-Seq-One, KA-Plus-Idem.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; (c + \neg c); \mathsf{id}$	Dist-L, Dist-R.
$\equiv$	$T_b\ a; n; c; r + a; n; \neg c; \mathsf{id} + \neg a; \mathsf{id}$	BA-Excl-Mid, KA-Seq-One.
$\equiv$	$T_b\ a; n; (c; r + \neg c; \mathsf{id}) + \neg a; \mathsf{id}$	Dist-L.
$\equiv$	T_b if a then n; if c then r else id else id	Sugar if statements.

□

Lemma 58 (Create an if with an unreachable true branch)**.** *For all predicates a and policies p and q, $a; p \equiv a;$ if a then p else q.*

Proof.

	Assertion	**Reasoning**
	$a; p$	
$\equiv$	$a; \mathsf{id}; p$	KA-Seq-One.
$\equiv$	$a; (a + \neg a); p$	BA-Excl-Mid.
$\equiv$	$a; (a; a + a; \neg a); p$	Dist-L.
$\equiv$	$a; (a; a + \mathsf{drop}); p$	BA-Contra.
$\equiv$	$a; (a; a; p + \mathsf{drop}; p)$	Dist-R.
$\equiv$	$a; (a; a; p + \mathsf{drop})$	KA-Zero-Seq.
$\equiv$	$a; (a; a; p + \mathsf{drop}; q)$	KA-Zero-Seq.
$\equiv$	$a; (a; a; p + a; \neg a; q)$	BA-Contra.
$\equiv$	$a; (a; p + \neg a; q)$	Dist-L, BA-Seq-Idem.
$\equiv$	$a;$ if a then p else q	Dist-L, BA-Seq-Idem.

□

Lemma 59 (Not false is true)**.** $\neg\mathsf{drop} \equiv \mathsf{id}$.

Proof.

	Assertion	**Reasoning**
	id	
$\equiv$	$\mathsf{drop} + \neg\mathsf{drop}$	BA-Excl-Mid.
$\equiv$	$\neg\mathsf{drop}$	KA-Plus-Drop.

□

Lemma 60 (Create an if with an unreachable false branch)**.** *For all predicates a and b and policies p and q, if $a;b \equiv \mathsf{drop}$ then $a;p \equiv a;\mathsf{if}\ b\ \mathsf{then}\ q\ \mathsf{else}\ p$.*

Proof.

	Assertion	**Reasoning**
	$a;p$	
$\equiv$	$a;\mathsf{id};p$	KA-Seq-One.
$\equiv$	$a;(\mathsf{drop}+\mathsf{id});p$	KA-Zero-Plus.
$\equiv$	$a;(\mathsf{drop};p+\mathsf{id};p)$	Dist-R.
$\equiv$	$a;(\mathsf{drop};q+\mathsf{id};p)$	KA-Zero-Seq.
$\equiv$	$a;(\mathsf{drop};q+\neg\mathsf{drop};p)$	Lemma 59.
$\equiv$	$a;(a;b;q+\neg a;b;p)$	Equals for equals.
$\equiv$	$a;(a;b;q+\neg a;p+\neg b;p)$	De Morgan's law, Dist-R.
$\equiv$	$(a;b;q+a;\neg a;p+a;\neg b;p)$	Dist-L, BA-Seq-Idem.
$\equiv$	$(a;b;q+\mathsf{drop}+a;\neg b;p)$	BA-Contra, KA-Seq-Zero.
$\equiv$	$(a;b;q+a;\neg b;p)$	KA-Plus-Zero.
$\equiv$	$a;(b;q+\neg b;p)$	Dist-L.
$\equiv$	$a;\mathsf{if}\ b\ \mathsf{then}\ q\ \mathsf{else}\ p$	Sugar if statement.

□

Lemma 61 (Multicast consolidation does not introduce tables where none previously existed)**.** *For all policies p and consolidation stages m and metadata annotations s, if p and m are closed (do not contain tables), and $m', n, \theta = \mathsf{pipeline}\ s\ p\ m$, then m' and n are closed and $\theta = \mathsf{id}$.*

Proof. The proof proceeds by a straightfoward induction on the structure of the policy p: only the cases where $p = (x : \tau)$ and $p = f \leftarrow v$ might produce policies with tables, but the former is ruled out by our hypotheses and the latter only introduces tables when a table already exists in the input consolidation stage, which is also ruled out by our hypotheses.

□

Lemma 62 (Substitution preserves typing)**.** *For all policies p and table bindings b and contexts Γ, if $p \vdash b\ \mathsf{wf}$ and $\Gamma \vdash p : \tau$, then $\vdash T_b\ p$.*

Proof. Induction on the structure of the typing derivation $\Gamma \vdash p : \tau$, relying on Definition 4 to replace tables with closed, similarly well-typed policies.

□

Lemma 63 (Types indicate sequential commutativity)**.** *For all policies p and q and contexts Γ_1, Γ_2 and well-formed table bindings b, if $\Gamma_1 \vdash p : (\mathsf{R}_p, \mathsf{W}_p)$ and $\Gamma_2 \vdash q : (\mathsf{R}_q, \mathsf{W}_q)$ and $\mathsf{W}_p \cap \mathsf{W}_q = \mathsf{W}_p \cap \mathsf{R}_q = \mathsf{W}_q \cap \mathsf{R}_p = \emptyset$, then $T_b\ p; q \equiv T_b\ q; p$.*

Proof. Follows from Lemma 5 and Lemma 4 and Lemma 62.

□

Theorem 9 (Multicast consolidation is semantics preserving)**.** *For all policies p, types $\tau_p = (\mathsf{R}_p, \mathsf{W}_p)$ and $\tau_m = (\mathsf{R}_m, \mathsf{W}_m)$, consolidation stages m_1, metadata annotations s, contexts Γ, and table bindings b, if*

1. *$p \vdash b$ wf and $m_1 \vdash b$ wf, and*
2. *$\Gamma \vdash p : \tau_p$ and $\Gamma \vdash m_1 : \tau_m$, and*
3. *$m_2, n, \theta = \mathsf{pipeline}\ s\ p\ m_1$,*

and let $z = \Pi_i f_i \leftarrow 0$, for all metadata fields f_i in the set of fresh metadata fields fs introduced in $\mathsf{pipeline}\ s\ p\ m_1$, then

1. *$m_2 \vdash \theta b$ wf and $n \vdash \theta b$ wf, and*
2. *there exists Γ' and Γ'' such that*
   - *$\Gamma' = \Gamma, \Gamma''$, and*
   - *$\Gamma' \vdash m_2 : (\mathsf{R}_m \cup \mathsf{R}_p \cup \mathsf{W}_p, \mathsf{W}_m \cup \{fs\})$, and*
   - *$\Gamma' \vdash n : (\mathsf{R}_p \cup fs, \mathsf{W}_p)$, and*
3. *$T_b\ z; p; m_1; z \equiv T_{\theta b}\ z; m_2; n; z$.*

Proof. The proof proceeds by mutual induction on the policy p and the consolidation stage m_1. The cases where $m_1 = \mathsf{drop}$ are immediate, as are the cases where the policy is id or drop.

A brief note on equivalence and binding substitution. In cases where binding substitutions are not applied directly, we omit the enclosing binding substitution to reduce the burden of notation. That is, we write $p \equiv q$ rather than $T_b\ p \equiv T_b\ q$.

Case $p = a$. For the first goal: $\theta = \mathsf{id}$, and so $\theta b = b$, and the result follows from H1. For the second goal: this case does not introduce new tables, so $\Gamma = \Gamma'$.

- By H2, $\Gamma \vdash p : (\mathsf{R}_p, \mathsf{W}_p)$ and $\Gamma \vdash m_1 \tau_m$, and by inversion $f \in \mathsf{R}_p \cup \mathsf{W}_p$. m_2 is equivalent to m_1 with $f = v$ distributed through the summation, and so $\Gamma \vdash m_2 : (\mathsf{R}_m \cup \mathsf{R}_p \cup \mathsf{R}_w, \mathsf{W}_m \cup \{fs\})$.
- $n = p$, and so $\Gamma \vdash n : (\mathsf{R}_p \cup fs, \mathsf{W}_p)$ follows from H2.

Finally, for the third goal: note that by definition, consolidation stages only modify fields that are fresh with respect to the original policy. Hence, the predicate a commutes with each atomic element of m and the result follows from [KAT-Commute].

Case $p = f \leftarrow v$. We have that $m_1 = m_1' + a; s$. The consolidation sequence can take one of two forms.

Subcase $s = \Pi_i f_i \leftarrow 1$. Note that $f \leftarrow v; (m_1' + a; s)$ distributes by [Dist-L] to $f \leftarrow v; m_1' + f \leftarrow v; a; s$, which is then equivalent by way of [PA-Mod-Filter] to $f \leftarrow v; m_1' + f \leftarrow v; f = v; a; s$. There are two cases for $f = v; a$: either it is equivalent to drop or not. In both cases, after unrolling $\mathsf{pipeline}$, we have that $(f = v; a') = \mathsf{specialize}\ (f = v\ a$ and $m_2', n_1, \theta_1 = \mathsf{pipeline}\ s\ f \leftarrow v\ m_1'$.

Sub-subcase $f = v; a \equiv \mathsf{drop}$. First, note that by Lemma 65 we have that $f = v; a \equiv f = va' \equiv \mathsf{drop}$. The results follow from the IH, using [KA-Zero-Seq], and [KA-Plus-Zero] for G3.

Sub-subcase $f = v; a \not\equiv \mathsf{drop}$**.** Again, note that by Lemma 65 we have that $f = v; a \equiv f = v; a'$. Furthermore, by the definition of $\mathsf{pipeline}$, n_1 is either $f \leftarrow v$ or drop, depending on whether m_1' is drop. The third goal holds by the following reasoning.

	Assertion	**Reasoning**
	$z; f \leftarrow v; (m_1' + a; s); z$	
$\equiv$	$z; f \leftarrow v; m_1'; z + z; f \leftarrow v; a; s; z$	[Dist-L, Dist-R]
$\equiv$	$z; m_2'; n_1; z + z; f \leftarrow v; a; s; z$	IH.
$\equiv$	$z; m_2'; n_1; z + z; f \leftarrow v; f = v; a; s; z$	[PA-Mod-Filter]
$\equiv$	$z; m_2'; n_1; z + z; f \leftarrow v; f = v; a'; s; z$	Lemma 65
$\equiv$	$z; m_2'; n_1; z + z; f \leftarrow v; a'; s; z$	[PA-Mod-Filter]
$\equiv$	$z; m_2'; n_1; z + z; a'; f \leftarrow v; s; z$	Lemma 63, Definition 9.
$\equiv$	$z; m_2'; n_1; z + z; a'; s; f \leftarrow v; z$	Definition of s.

Hence, if $n_1 = \mathsf{drop}$ then m_2' is drop and the result is $z; (m_2' + a'; s); f \leftarrow v; z$ by KA-Plus-Zero. Otherwise, if $n_1 = f \leftarrow v$, then the same result follows from distribution.

The first goal holds as a result of the IH. The second follows from the IH and Lemma 65.

Subcase $s = \Pi_i f_i \leftarrow 1; (x : \tau)$**.** Let $q = f_i \leftarrow 1$. After unfolding $\mathsf{pipeline}$, we have the following.

$$
\begin{array}{lcl}
\theta_1 & = & (\lambda\ b, y. \\
 & & \quad \mathsf{let}\ m', \_, \_ = \mathsf{pipeline}\ s\ f \leftarrow v\ (T_b\ a; (x : \tau); q)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ y = z\ \mathsf{then}\ m' \\
 & & \quad \mathsf{else} T_b\ y) \\
\tau_2 & = & \mathsf{typeof}\ (a; q; (x : \tau)) \\
m_2, \_, \theta_2 & = & \mathsf{pipeline}\ s\ f \leftarrow v\ m_1'
\end{array}
$$

Let $z = z_1; z_2$, where z_1 and z_2 zero metadata introduced in the recursive calls to pipeline. From the IH, we also have the following.

$$\begin{aligned} T_{\mathsf{id}}\ z_1; f \leftarrow v; (T_b\ a; (x : \tau); q); z_1 &\equiv T_{\mathsf{id}}\ z_1; m'; n'; z_1 \\ T_b\ z_2; f \leftarrow v; m'_1; z_2 &\equiv T_{\theta_2 b}\ z_2; m_2; f \leftarrow v; z_2 \end{aligned}$$

For the third goal, we must show the following.

$$T_b\ z; f \leftarrow v; (m'_1 + a; q; (x : \tau)); z \equiv T_{\theta_1 \circ \theta_2 b}\ z; (m_2 + (z : \tau_2)); f \leftarrow v; z$$

Before beginning, note that by construction θ_1 and θ_2 introduce bindings for different tables, and so from Lemma 68 we have that $T_{\theta_1 \circ \theta_2 b}\ m_2 = T_{\theta_2 b}\ m_2$ and $T_{\theta_1 \circ \theta_2 b}\ (z : \tau_2) = T_{\theta_1 b}\ (z : \tau_2)$.

	Assertion	**Reasoning**
	$T_b\ z; f \leftarrow v; (m'_1 + a; q; (x : \tau)); z$	
$\equiv$	$T_b\ z; f \leftarrow v; m'_1; z + z; f \leftarrow v; a; (x : \tau); q; z$	Dist-L, Dist-R.
$\equiv$	$T_b\ z_1; z_2; f \leftarrow v; m'_1; z_2; z_1 + z_2; z_1; f \leftarrow v; a; (x : \tau); q; z_1; z_2$	PA-Mod-Comm.
$\equiv$	$z_1; T_b\ (z_2; f \leftarrow v; m'_1; z_2); z_1$	
	$+ z_2; T_{\mathsf{id}}\ (z_1; f \leftarrow v; (T_b\ a; (x : \tau); q); z_1); z_2$	Definition of T.
$\equiv$	$z_1; T_{\theta_2 b}\ (z_2; m_2; f \leftarrow v; z_2); z_1 + z_2; T_{\mathsf{id}}\ (z_1; m'; f \leftarrow v; z_1); z_2$	IH.
$\equiv$	$z_1; z_2; (T_{\theta_2 b}\ m_2); f \leftarrow v; z_2; z_1 + z_2; z_1; m'; f \leftarrow v; z_1; z_2$	Definition of T.
$\equiv$	$z; (T_{\theta_2 b}\ m_2); f \leftarrow v; z + z; m'; f \leftarrow v; z$	Equals for equals, PA-Mod-Comm.
$\equiv$	$z; (T_{\theta_2 b}\ m_2 + m'); f \leftarrow v; z$	Dist-L, Dist-R.
$\equiv$	$z; (T_{\theta_2 b}\ m_2 + T_{\theta_1 b}\ (z : \tau_2)); f \leftarrow v; z$	Definition of θ_1.
$\equiv$	$z; (T_{\theta_1 \circ \theta_2 b}\ m_2 + T_{\theta_1 \circ \theta_2 b}\ (z : \tau_2)); f \leftarrow v; z$	Equals for equals.
$\equiv$	$T_{\theta_1 \circ \theta_2 b}\ z; (m_2 + (z : \tau_2)); f \leftarrow v; z$	Definition of T.

The first goal follows from H1, the IH, and the fact that invoking pipeline on closed policies produces closed policies (Lemma 61). The second goal follows from H2.

Case $p = x : (\mathsf{R}, \mathsf{W})$**.** After unrolling pipeline, we have the following.

$$
\begin{array}{lcl}
\tau & = & \mathsf{typeof}\ m \\
t_m & = & y : (\mathsf{R}, fs) \cup \tau \\
t_n & = & z : (\mathsf{R} \cup fs, \mathsf{W}) \\
\theta' & = & (\lambda b, w. \\
 & & \quad \mathsf{let}\ m', n, \theta = \mathsf{pipeline}\ s\ (T_b\ x)\ (T_b\ m_1)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ w = y\ \mathsf{then}\ m' \\
 & & \quad \mathsf{else\ if}\ w = z\ \mathsf{then}\ n \\
 & & \quad \mathsf{else}\ T_{\theta\ b}\ w)
\end{array}
$$

And from the IH, we have the following.

$$T_b\ z; (x : (\mathsf{R}, \mathsf{W})); m; z \equiv T_{\theta' b}\ z; t_m; t_n; z$$

	Assertion	**Reasoning**
	$T_b\ z; (x : (\mathsf{R}, \mathsf{W})); m; z$	
$\equiv$	$T\mathsf{id}z; T_b\ ((x : (\mathsf{R}, \mathsf{W})); m); z$	Definition of T.
$\equiv$	$T\mathsf{id}z; m'; n; z$	IH.
$\equiv$	$z; m'; n; z$	Definition of T.
$\equiv$	$T_{\theta' b}\ z; t_m; t_n; z$	Definition of θ'.

The first goal follows from Lemma 61 and from H1. The second goal follows from H2 and the definitions of t_m and t_n.

Case $p = \mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$**.** After unfolding pipeline, we have the following.

$$
\begin{array}{lcl}
\Sigma_i a_i; s_i, n_1, \theta_1 & = & \mathsf{pipeline}\ s\ p_1\ m \\
\Sigma_j a_j; s_j, n_2, \theta_2 & = & \mathsf{pipeline}\ s\ p_2\ m
\end{array}
$$

Let $z = z_1; z_2$, where z_1 and z_2 zero the metadata introduced in the recursive calls to pipeline. From the IH, we have the following.

$$\begin{aligned} z_1; p_1; m; z_1 &\equiv z_1; (\Sigma_i a_i; s_i); n_1; z_1 \\ z_2; p_2; m; z_2 &\equiv z_2; (\Sigma_j a_j; s_j); n_2; z_2 \end{aligned}$$

	Assertion	**Reasoning**
	$z;$ if b then p_1 else $p_2; m; z$	
$\equiv$	$z; (b; p_1 + \neg b; p_2); m; z$	Desugar if statement.
$\equiv$	$z; b; p_1; m; z + z; \neg b; p_2; m; z$	Dist-L, Dist-R.
$\equiv$	$z_1; z_2; b; p_1; m; z_1; z_2 + z_1; z_2; \neg b; p_2; m; z_1; z_2$	Equals for equals.
$\equiv$	$z_2; z_1; b; p_1; m; z_1; z_2 + z_1; z_2; \neg b; p_2; m; z_2; z_1$	PA-Mod-Mod-Comm.
$\equiv$	$z_2; b; z_1; p_1; m; z_1; z_2 + z_1; \neg b; z_2; p_2; m; z_2; z_1$	KAT-Commutes, freshness.
$\equiv$	$z_2; b; z_1; (\Sigma_i a_i; s_i); n_1; z_1; z_2 + z_1; \neg b; z_2; (\Sigma_j a_j; s_j); n_2; z_2; z_1$	IH.
$\equiv$	$z_2; z_1; b; (\Sigma_i a_i; s_i); n_1; z_1; z_2 + z_1; z_2; \neg b; (\Sigma_j a_j; s_j); n_2; z_2; z_1$	KAT-Commutes, freshness.
$\equiv$	$z_2; z_1; (\Sigma_i b; a_i; s_i); n_1; z_1; z_2 + z_1; z_2; (\Sigma_j \neg b; a_j; s_j); n_2; z_2; z_1$	Dist-L.
$\equiv$	$z_1; z_2; (\Sigma_i b; a_i; s_i); n_1; z_1; z_2 + z_1; z_2; (\Sigma_j \neg b; a_j; s_j); n_2; z_1; z_2$	PA-Mod-Mod-Comm.
$\equiv$	$z; (\Sigma_i b; a_i; s_i); n_1; z + z; (\Sigma_j \neg b; a_j; s_j); n_2; z$	Equals for equals.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); n_1 + (\Sigma_j \neg b; a_j; s_j); n_2); z$	Dist-L, Dist-R.
$\equiv$	$z; ((\Sigma_i b; b; a_i; s_i); n_1 + (\Sigma_j \neg b; \neg b; a_j; s_j); n_2); z$	BA-Seq-Idem.
$\equiv$	$z; ((\Sigma_i b; a_i; b; s_i); n_1 + (\Sigma_j \neg b; a_j; \neg b; s_j); n_2); z$	BA-Seq-Comm.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i; b); n_1 + (\Sigma_j \neg b; a_j; s_j; \neg b); n_2); z$	KAT-Commutes, freshness.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); b; n_1 + (\Sigma_j \neg b; a_j; s_j); \neg b; n_2); z$	Dist-R.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); b; n_1 + \mathsf{drop} + \mathsf{drop} + (\Sigma_j \neg b; a_j; s_j); \neg b; n_2); z$	KA-Plus-Zero.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); b; n_1 + (\Sigma_j \neg b; a_j; s_j); \mathsf{drop}; n_1; +$ $(\Sigma_i b; a_i; s_i); \mathsf{drop}; n_2 + (\Sigma_j \neg b; a_j; s_j); \neg b; n_2); z$	KA-Seq-Zero, KA-Zero-Seq.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); b; n_1 + (\Sigma_j \neg b; a_j; s_j); \neg b; b; n_1; +$ $(\Sigma_i b; a_i; s_i); b; \neg b; n_2 + (\Sigma_j \neg b; a_j; s_j); \neg b; n_2); z$	BA-Contra.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i); b; n_1 + (\Sigma_j \neg b; a_j; s_j); b; n_1; +$ $(\Sigma_i b; a_i; s_i); \neg b; n_2 + (\Sigma_j \neg b; a_j; s_j); \neg b; n_2); z$	BA-Seq-Idem, BA-Seq-Comm, KAT-Commutes, freshness.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i) + (\Sigma_j \neg b; a_j; s_j)); (b; n_1 + \neg b; n_2); z$	Dist-L, Dist-R.
$\equiv$	$z; ((\Sigma_i b; a_i; s_i) + (\Sigma_j \neg b; a_j; s_j));$ if b then n_1 else $n_2; z$	Sugar if statements.

The first goal follows from the IH. The second goal follows from the IH as well, observing that adding b to the consolidation stage types under $\mathsf{R}_p \cup \mathsf{W}_p$. Finally, the third goal follows from if statement de-nesting (Lemma 70) and qualification (Lemma **??**):

$$\mathsf{if}\ b\ \mathsf{then}\ n_1\ \mathsf{else}\ n_2 \equiv \mathsf{qualify}\ b\ n_1; \mathsf{qualify}\ b\ n_2$$

Case $p = p_1 + p_2$. After unrolling pipeline, we have the following.

$$\begin{array}{lcl} m_1', n_1, \theta_1 & = & \mathsf{pipeline}\ s\ p_1\ m \\ \Sigma_j a_j; s_j, n_2, \theta_2 & = & \mathsf{pipeline}\ s\ p_2\ m \\ n_1' & = & \mathsf{qualify}\ f = 0\ n_1 \\ n_2' & = & \mathsf{qualify}\ f = 1\ n_2 \end{array}$$

Let $z = f \leftarrow 0; z_1; z_2$, where z_1 and z_2 zero the metadata introduced in the recursive calls to pipeline. From the IH, we also have the following.

$$\begin{array}{lcl} z_1; p_1; m; z_1 & \equiv & z_1; m_1'; n_1; z_1 \\ z_2; p_2; m; z_2 & \equiv & z_2; (\Sigma_j a_j; s_j); n_2; z_2 \end{array}$$

Lemma 57 also leads to the following.

$$\begin{array}{lcl} n_1' & \equiv & \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id} \\ n_2' & \equiv & \mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id} \end{array}$$

With these facts in mind, we can show the third goal as follows.

	Assertion	**Reasoning**
	$z; (p_1 + p_2); m; z$	
$\equiv$	$z; p_1; m; z + z; p_2; m; z$	Dist-L, Dist-R.
$\equiv$	$f \leftarrow 0; z_2; z_1; p_1; m; z_1; z_2; f \leftarrow 0 +$	
	$f \leftarrow 0; z_1; z_2; p_2; m; z_2; f \leftarrow 0; z_1$	Equals for equals, PA-Mod-Mod-Comm.

$\equiv$	$f \leftarrow 0; z_2; z_1; m_1'; n_1; z_1; z_2; f \leftarrow 0 +$ $f \leftarrow 0; z_1; z_2; (\Sigma_j a_j; s_j); n_2; z_2; f \leftarrow 0; z_1$	IH.
$\equiv$	$z; m_1'; n_1; z + z; (\Sigma_j a_j; s_j); n_2; z$	Equals for equals, PA-Mod-Mod-Comm.
$\equiv$	$z; (m_1'; n_1 + (\Sigma_j a_j; s_j); n_2); z$	Dist-L, Dist-R.
$\equiv$	$z; f = 0; (m_1'; n_1 + (\Sigma_j a_j; s_j); n_2); z$	PA-Mod-Filter, PA-Mod-Filter-Comm.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; s_j); n_2); z$	KAT-Commute, freshness.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; s_j); n_2); f \leftarrow 0; z$	PA-Mod-Mod-Comm, PA-Mod-Filter, PA-Filter-Mod.
$\equiv$	$z; (m_1'; f = 0; n_1; f \leftarrow 0 + f = 0; (\Sigma_j a_j; f \leftarrow 0; s_j); n_2); z$	KAT-Commute, freshness, Dist-R.
$\equiv$	$z; (m_1'; f = 0; n_1; f \leftarrow 0 +$ $f = 0; (\Sigma_j a_j; f \leftarrow 1; f \leftarrow 0; s_j); n_2); z$	PA-Mod-Mod.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j); n_2); f \leftarrow 0; z$	Dist-R, freshness, KAT-Commute.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j); n_2); z$	PA-Mod-Mod-Comm, PA-Mod-Filter, PA-Filter-Mod.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; f \leftarrow 1; f = 1; s_j); n_2); z$	PA-Mod-Filter.
$\equiv$	$z; (m_1'; f = 0; n_1 + f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j); f = 1; n_2); z$	KAT-Commute, freshness, Dist-R.
$\equiv$	$z; (m_1'; f = 0; \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id} +$ $f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j);$ $f = 1; \mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id}); z$	Lemma 58.
$\equiv$	$z; (m_1'; f = 0; \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id};$ $\mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id} +$ $f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j);$ $f = 1; \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id};$ $\mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id}); z$	Lemma 60.
$\equiv$	$z; (m_1'; f = 0 + f = 0; (\Sigma_j a_j; f \leftarrow 1; s_j); f = 1);$ $\mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id}; \mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id}); z$	Dist-R.
$\equiv$	$z; (m_1' + (\Sigma_j a_j; f \leftarrow 1; s_j);$ $f = 1); \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id};$ $\mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id}); z$	KAT-Commute, freshness, Dist-L.
$\equiv$	$z; (m_1' + (\Sigma_j a_j; f \leftarrow 1; s_j)); \mathsf{if}\ f = 0\ \mathsf{then}\ n_1\ \mathsf{else}\ \mathsf{id};$ $\mathsf{if}\ f = 1\ \mathsf{then}\ n_2\ \mathsf{else}\ \mathsf{id}); z$	KAT-Commute, freshness, Dist-R, PA-Mod-Filter.

$\equiv \quad z; (m_1' + \Sigma_j; a_j; f \leftarrow 1; s_j); n_1'; n_2'; z \quad \mid$ Lemma 57.

The first goal follows from the IH. The second follows from the IH and H2.

Case $p = p_1; p_2$**.** After unrolling pipeline, we have the following.

$$\begin{aligned} m_2, n_2, \theta_2 &= \mathsf{pipeline}\ s\ p_2\ m \\ m_1, n_1, \theta_1 &= \mathsf{pipeline}\ s\ p_1\ m_2 \end{aligned}$$

Applying the IH yields the following.

$$\begin{aligned} z_2; p_2; m; z_2 &\equiv z_2; m_2; n_2; z_2 \\ z_1; p_1; m_2; z &\equiv z; m_1; n_1; z_1 \end{aligned}$$

We must show the following.

$$z; p_1; p_2; m; z \equiv z; m_1; n_1; n_2; z$$

Let $z = z_1; z_2$, where z_1 and z_2 zero metadat introduced in the recursive calls to pipeline.

	Assertion	**Reasoning**
	$z; p_1; p_2; m; z$	
$\equiv$	$z_1; z_2; p_1; p_2; m; z_1; z_2$	Equals for equals.
$\equiv$	$z_1; p_1; z_2; p_2; m; z_1; z_2$	KAT-Commute, freshness.
$\equiv$	$z_1; p_1; z_2; p_2; m; z_2; z_1$	[PA-Mod-Comm]
$\equiv$	$z_1; p_1; z_2; m_2; n_2; z_2; z_1$	IH.
$\equiv$	$z_2; z_1; p_1; m_2; n_2; z_1; z_2$	[PA-Mod-Comm]
$\equiv$	$z_2; z_1; p_1; m_2; z_1; n_2; z_2$	KAT-Commute, freshness.
$\equiv$	$z_2; z_1; m_1; n_1; z_1; n_2; z_2$	IH.
$\equiv$	$z_2; z_1; m_1; n_1; n_2; z_1; z_2$	KAT-Commute, freshness.
$\equiv$	$z_1; z_2; m_1; n_1; n_2; z_1; z_2$	[PA-Mod-Comm]
$\equiv$	$z; m_1; n_1; n_2; z$	Equals for equals.

The first and second goals follow from the IH.

Case $p = p_1 \ \mathsf{W}_1||_{\mathsf{W}_2} \ p_2$**.** Follows from the case where $p = p_1; p_2$ and Lemma 5.

□

C.2.3 Refactoring Field Modification

The packet processing pipeline on the RMT chip is conceptually divided into three parts: an initial pipeline of ingress match stages, a set of queues, and a final pipeline of egress match stages. Multicast is implemented by copying pointers to the packet into queues. Each queue is associated with a given output port, and the number of times a packet pointer is enqueued corresponds to the number of copies emitted from that port. Optionally, each packet pointer may also have an identifier set for individualized processing in the egress pipeline. In order to compile our (r, m, n) structured policy to the RMT architecture, we need to know which output port a packet copy is destined for at the time the packet is copied (*i.e.* in m).

Lemma 64. *For all predicates c, policies p_1, p_2, q_1, q_2, z, and fresh fields f, if $z \equiv f \leftarrow 0; \Pi_i f_i \leftarrow 0$, then*

$$z; \mathsf{if}\ c\ \mathsf{then}\ p_1; p_2\ \mathsf{else}\ q_1; q_2; z \equiv z; (\mathsf{if}\ c\ \mathsf{then}\ p_1; f \leftarrow 1\ \mathsf{else}\ q_1); \mathsf{if}\ f = 1\ \mathsf{then}\ p_2\ \mathsf{else}\ q_2; z$$

Proof.

	Assertion	**Reasoning**
	$z; \mathsf{if}\ c\ \mathsf{then}\ p_1; p_2\ \mathsf{else}\ q_1; q_2; z$	
$\equiv$	$z; c; p_1; p_2; z + z; \neg c; q_1; q_2; z$	[Desugar]
$\equiv$	$z; c; p_1; p_2; z + \mathsf{drop}$	
$+$	$\mathsf{drop} + z; \neg c; q_1; q_2; z$	[Plus-Zero]
$\equiv$	$z; c; p_1; p_2; z + \mathsf{drop}$	
$+$	$\mathsf{drop} + z; \neg c; q_1; \neg f = 1; q_2; z$	[Freshness, z, BA-Inverse]
$\equiv$	$z; c; p_1; p_2; z + \mathsf{drop}$	
$+$	$z; c; p_1; f \leftarrow 1; \neg f = 1; q_2; z + z; \neg c; q_1; \neg f = 1; q_2; z$	[BA-Contra]
$\equiv$	$z; c; p_1; p_2; z + z; \neg c; q_1; f = 1; p_2; z$	
$+$	$z; c; p_1; f \leftarrow 1; \neg f = 1; q_2; z + z; \neg c; q_1; \neg f = 1; q_2; z$	[Freshness, z]
$\equiv$	$z; c; p_1; f \leftarrow 1; p_2; z + z; \neg c; q_1; f = 1; p_2; z$	
$+$	$z; c; p_1; f \leftarrow 1; \neg f = 1; q_2; z + z; \neg c; q_1; \neg f = 1; q_2; z$	[Freshness, z]
$\equiv$	$z; c; p_1; f \leftarrow 1; f = 1; p_2; z + z; \neg c; q_1; f = 1; p_2; z$	
$+$	$z; c; p_1; f \leftarrow 1; \neg f = 1; q_2; z + z; \neg c; q_1; \neg f = 1; q_2; z$	[Mod-Match]
$\equiv$	$z; (c; p_1; f \leftarrow 1 + \neg c; q_1); (f = 1; p_2 + \neg f = 1; q_2); z$	
$\equiv$	$z; (c; p_1; f \leftarrow 1 + \neg c; q_1); \mathsf{if}\ f = 1\ \mathsf{then}\ p_2\ \mathsf{else}\ q_2; z$	[Sugar]

□

Lemma 65 (Specialization is semantics preserving)**.** *For all fields f, values v, and predicates a, $f = v; a \equiv \mathsf{specialize}\ f = v\ a$, and if $\vdash a : \tau$ then $\vdash a' : \tau$.*

Proof. This lemma is a straightforward generalization of Lemma 32. The typing result follows from a straightforward induction on the typing derivation.

□

Lemma 66 (Sequenced extraction of closed policies is semantics preserving)**.** *For all multicast-free policies r, and modification stages e, and fields f, and metadata estimates s, if*

1. $\vdash r : (\mathsf{R}_r, \mathsf{W}_r)$, *and*
2. $\vdash e : (\mathsf{R}_e, \mathsf{W}_e)$, *and*
3. $\mathsf{R}_r \cap \mathsf{W}_e = \{f\}$ *and* $\mathsf{W}_r \cap \mathsf{W}_e = \{f\}$, *and*
4. $(e', r', \theta) = \mathsf{ext}_f\ s\ r\ e$, *and*
5. $\mathsf{ext}_f\ s\ r\ e$ *produces fresh metadata fields fs, and*
6. r *and* e *are closed,*

then

- *let* $z = \Pi_{f \in fs} f \leftarrow 0$,
- $\vdash r' : (\mathsf{R}_r, \mathsf{W}_r \setminus \{f\})$, *and*
- $\vdash e' : (\mathsf{R}_e \cup \mathsf{R}_r, \mathsf{W}_e \cup \{f\} \cup fs)$, *and*
- r' *and* e' *are closed, and*
- $\theta = \mathsf{id}$, *and*

- $z; r; e; z \equiv z; e'; r'; z$.

Proof. The proof proceeds by mutual induction on r and e. The base case of e, where $e = \mathsf{drop}$, is always immediate, leaving $e = e_1 + a; m$. Aside from modification, the base cases of r are immediate (or contradictory, in the case where r is a table). We examine the remaining cases below.

Case $r = f' \leftarrow v$. We have that $e = e_1 + a; \Pi_i f_i \leftarrow v_i$, as the alternative—where e includes a table—is contradictory. Reducing the application of $\mathsf{ext}_f\ s\ f' \leftarrow v\ e$ (Definition 15) yields the following.

$$
\begin{array}{l}
\mathsf{let}\ (f' = v; a') = \mathsf{specialize}\ (f' = v)\ a\ \mathsf{in} \\
\mathsf{let}\ e', p', \theta = \mathsf{ext}_f\ (f' \leftarrow v)\ e\ \ \mathsf{in} \\
\mathsf{if}\ (f' = v; a') \equiv \mathsf{drop}\ \mathsf{then} \\
\quad (e', p', \theta) \\
\mathsf{else\ if}\ f = f'\ \mathsf{then} \\
\quad ((e' + a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i), \mathsf{id}, \theta) \\
\mathsf{else} \\
\quad ((e' + a'; \Pi_i f_i \leftarrow v_i), f' \leftarrow v, \theta)
\end{array}
$$

Intuitively, the goal of this function is to compute some e' and p' such that $p; e \equiv e'; p'$ and modifications of f in p are moved to e' and replaced with id in p'. From the induction hypothesis, we have that

$$z; f' \leftarrow v; e_1; z \equiv z; e'; p'; z$$

There are three cases.

Subcase $f' = v; a' \equiv \mathsf{drop}$. We must show the following.

$$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z \equiv z; e'; p'; z$$

	Assertion	Reasoning
	$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z$	
$\equiv$	$z; f' \leftarrow v; e_1; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[Dist-L, Dist-R]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[IH]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; f' = v; a; \Pi_i f_i \leftarrow v_i; z$	[Mod-Filter]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; \mathsf{drop}; \Pi_i f_i \leftarrow v_i; z$	[Lemma 65, case assumption]
$\equiv$	$z; e'; p'; z$	[Seq-Zero, Zero-Seq]

Subcase $f' = v; a' \not\equiv \mathsf{drop}$ and $f = f'$. We must show the following.

$$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z \equiv z; (e' + a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i); \mathsf{id}; z$$

First, note from the definition of ext , $\mathrm{p}' = \mathsf{id}$ in this case.

	Assertion	Reasoning
	$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z$	
$\equiv$	$z; f' \leftarrow v; e_1; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[Dist-L, Dist-R]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[IH]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; f' = v; a'; \Pi_i f_i \leftarrow v_i; z$	[Mod-Filter, Lemma 65]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; a'; \Pi_i f_i \leftarrow v_i; z$	[Mod-Filter]
$\equiv$	$z; e'; p'; z+$	
	$z; a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i; z$	[Lemma 4.4 [3], Definition 9]
$\equiv$	$z; e'; \mathsf{id}; z+$	
	$z; a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i; \mathsf{id}; z$	[Seq-One, $p' = \mathsf{id}$]
$\equiv$	$z; (e' + a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i); \mathsf{id}; z$	[Dist-R, Dist-L]

Subcase $f' = v; a' \not\equiv \mathsf{drop}$ and $f \neq f'$. We must show the following. First, note from the definition of ext , $\mathrm{p}' = f' \leftarrow v$ in this case.

$$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z \equiv z; (e' + a'; \Pi_i f_i \leftarrow v_i); f' \leftarrow v; z$$

	Assertion	**Reasoning**
	$z; f' \leftarrow v; (e_1 + a; \Pi_i f_i \leftarrow v_i); z$	
$\equiv$	$z; f' \leftarrow v; e_1; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[Dist-L, Dist-R]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; a; \Pi_i f_i \leftarrow v_i; z$	[IH]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; f' = v; a'; \Pi_i f_i \leftarrow v_i; z$	[Mod-Filter, Lemma 65]
$\equiv$	$z; e'; p'; z+$	
	$z; f' \leftarrow v; a'; \Pi_i f_i \leftarrow v_i; z$	[Mod-Filter]
$\equiv$	$z; e'; p'; z+$	
	$z; a'; f' \leftarrow v; \Pi_i f_i \leftarrow v_i; z$	[Lemma 4.4 [3], Definition 9]
$\equiv$	$z; e'; p'; z+$	
	$z; a'; \Pi_i f_i \leftarrow v_i; f' \leftarrow v; z$	[Mod-Mod-Comm, case assumption, hypothesis $\mathsf{W}_e \cap \mathsf{W}_r \subseteq \{f\}$]
$\equiv$	$z; e'; f' \leftarrow v; z+$	
	$z; a'; \Pi_i f_i \leftarrow v_i; f' \leftarrow v; z$	$[p' = f' \leftarrow v]$
$\equiv$	$z; (e' + a'; \Pi_i f_i \leftarrow v_i); f' \leftarrow v; z$	[Dist-R, Dist-L]

In each subcase, the other goals (besides equivalence) immediately hold.

Case $r = \mathsf{if}\ b\ \mathsf{then}\ r_1\ \mathsf{else}\ r_2$. From the reduction of $\mathsf{ext}_f\ s\ (\mathsf{if}\ b\ \mathsf{then}\ r_1\ \mathsf{else}\ r_2)\ e$, we have

- $(\sum_i a_{1i}, m_{1i}), p_1, \theta_1 = \mathsf{ext}_f\ s\ r_1\ e$
- $(\sum_j a_{2j}, m_{2j}), q_2, \theta_2 = \mathsf{ext}_f\ s\ r_2\ e$
- $e' = \sum_i b; a_{1i}; m_{1i}; f' \leftarrow 1 + \sum_j \neg b; a_{2j}; m_{2j}$
- $r' = \mathsf{if}\ f' = 1\ \mathsf{then}\ p_1\ \mathsf{else}\ q_2$
- $\theta = \theta_2 \circ \theta_1$

From the induction hypothesis, it follows that

- $(\sum_i a_{1i}; m_{1i}); r_{11} \equiv r_1; e$
- $(\sum_j a_{2j}; m_{2j}); r_{12} \equiv r_2; e$

We must show that

$$z; \mathsf{if}\ b\ \mathsf{then}\ r_1\ \mathsf{else}\ r_2; e; z \equiv z; e'; r'; z$$

The equivalence result follows from instantiating Lemma 64 with

- $c = b$
- $p_1 = (\sum_i a_{1i}; m_{1i})$
- $p_2 = r_{11}$
- $q_1 = (\sum_j a_{2j}; m_{2j})$
- $q_2 = r_{12}$

and substituting equals for equals. The other goals are immediate.

Case $r = r_1; r_2$**.** After reducing $\mathsf{ext}_f\ s\ r_1; r_2\ e$, we have

- $e_1, r_4, \theta_1 = \mathsf{ext}_f\ s\ r_2\ e$
- $e_2, r_3, \theta_2 = \mathsf{ext}_f\ s\ r_1\ e_1$

From the induction hypothesis, we have

- $z \equiv z_1; z_2$
- $z_1; r_2; e; z_1 \equiv z_1; e_1; r_4; z_1$
- $z_2; r_1; e_1; z_2 \equiv z_2; e_2; r_3; z_2$

We must show

$$z; r_1; r_2; e; z \equiv z; e_2; r_3; r_4; z$$

	Assertion	Reasoning
	$z; r_1; r_2; e; z$	
$\equiv$	$z_1; z_2; r_1; z; r_2; e; z_1; z_2$	[Substitution]
$\equiv$	$z_2; r_1; z_1; r_2; e; z_1; z_2$	[Freshness]
$\equiv$	$z_2; r_1; z_1; e_1; r_4; z_1; z_2$	[Substitution]
$\equiv$	$z_1; z_2; r_1; e_1; r_4; z_1; z_2$	[Freshness]
$\equiv$	$z_1; z_2; r_1; e_1; z_2; r_4; z_1$	[Freshness]
$\equiv$	$z_1; z_2; e_2; r_3; z_2; r_4; z_1$	[Substitution]
$\equiv$	$z_1; z_2; e_2; r_3; r_4; z_1; z_2$	[Freshness]
$\equiv$	$z; e_2; r_3; r_4; z$	[Freshness]

Case $r = r_1 \,||\, r_2$**.** After rewriting with [Con-Seq], this case proceeds identically as the case where $r = r_1; r_2$.

□

Lemma 67 (Compilation extends table bindings)**.** *For all policies p, r, e, fields f, table bindings b, if*

- $e', r', \theta = \mathsf{ext}_f\ r\ e$ *, and*
- $\mathsf{ext}_f\ r\ e$ *introduces tables $x_1 : \tau_1, \ldots, x_n : \tau_n \in xs$,*

then for all policies p that do not contain tables xs, $T_b\ p = T_{\theta\ b}\ p$.

Proof. Induction on the structure of r and the definition of T . □

Lemma 68 (Transforming a pre-compiled policy has no effect)**.** *For all policies p, table bindings b, and binding transformers θ, if*

- *let xs be the tables that appear in p*
- *$T_b\ p$ produces a closed term, and*
- *for all $x : \tau \in xs$, $T_b\ x = T_{\theta\ b}\ x$*

then $T_b\ p = T_{\theta\ b}\ p$.

Proof. Induction on p. □

Theorem 10 (Field extraction is semantics preserving)**.** *For all multicast-free policies r, modification stages e, table bindings b, contexts Γ, and fields f, if*

1. $r \vdash b$ wf *and* $e \vdash b$ wf*, and*
2. $\Gamma \vdash r : (\mathsf{R}_r, \mathsf{W}_r)$*, and* $\Gamma \vdash e : (\mathsf{R}_e, \mathsf{W}_e)$*, and*
3. $\mathsf{R}_r \cap \mathsf{W}_e = \{f\}$ *and* $\mathsf{W}_r \cap \mathsf{W}_e = \{f\}$*, and*
4. $(e'; r', \theta) = \mathsf{ext}_f\ s\ r\ e$*, and*

and let $z = \Pi_i f_i \leftarrow 0$, for all metadata fields f_i in the set of fresh metadata fields fs introduced in $\mathsf{ext}_f\ s\ r\ e$*, then*

1. $e' \vdash \theta b$ wf *and* $r' \vdash \theta b$ wf*, and*
2. *there exists Γ' and Γ'' such that*
   - $\Gamma' = \Gamma, \Gamma''$*, and*
   - $\Gamma' \vdash r' : (\mathsf{R}_r, \mathsf{W}_r \setminus \{f\})$*, and*
   - $\Gamma' \vdash e' : (\mathsf{R}_e \cup \mathsf{R}_r, \mathsf{W}_e \cup \{f\} \cup fs)$*, and*
3. $T_b\ (z; r; e; z) \equiv T_{\theta\ b}\ (z; e'; r'; z)$.

Proof. Mutual induction on e and r. The base case of e is immediate, leaving $e = (e + a; m)$, and the cases for $r = \mathsf{id}$ and $r = \mathsf{drop}$ are immediate, and $r = a$ follows from the induction hypothesis. The combinator cases proceed similarly to Lemma 66, relying on Lemmas 67 and 68 to reason about composing binding transformers. The interesting cases are modifications and table variables.

Case $r = f' \leftarrow v, e = e_0 + a; m$**.** When m does not contain a table, the result follows from the induction hypothesis, similar to Lemma 66. Otherwise, $m = a; q; (x : \tau)$.

After unrolling, we have:

$$
\begin{array}{l}
\mathsf{let}\ e_1 = a; q; (x : \tau)\ \mathsf{in} \\
\mathsf{let}\ \tau_1 = \mathsf{typeof}\ e_1\ \mathsf{in} \\
\mathsf{let}\ p_1 = x_1 : \tau_1 \cup (\emptyset, \{f\})\ \mathsf{in} \\
\mathsf{let}\ \theta_1 = (\lambda b, y. \\
\quad \mathsf{let}\ e_2, p_2, \theta_2 = \\
\qquad \mathsf{ext}_f\ s\ (f \leftarrow v)\ (T_b\ e_1) \\
\quad \mathsf{if}\ y = x_1\ \mathsf{then}\ e_2 \\
\quad \mathsf{else}\ T_b\ y)\ \mathsf{in} \\
\mathsf{let}\ e_2, \_, \theta_3 = \mathsf{ext}_f\ s\ (f \leftarrow v)\ e_0\ \mathsf{in} \\
\mathsf{if}\ f = f'\ \mathsf{then} \\
\quad (e_2 + p_1, \mathsf{id}, \theta_3 \circ \theta_1) \\
\mathsf{else} \\
\quad (e_2 + p_1, f' \leftarrow v, \theta_3 \circ \theta_1)
\end{array}
$$

There are two cases.

Subcase $f = f'$. After reducing $\mathsf{ext}_f\ s\ f' \leftarrow v\ e$ when $f = f'$, we must show the following.

$$T_b\ (z; f \leftarrow v; (e_0 + a; q; (x : \tau)); z) \equiv T_{(\theta_3 \circ \theta_1)\ b}\ (z; (e_2 + p_1); \mathsf{id}; z)$$

From the IH, we have

$$T_b\ (z; f \leftarrow v; e_0; z) \equiv T_{\theta_3 b}\ (z; e_2; \mathsf{id}; z)$$

And from Lemma 66, we have

$$z; e_2; z \equiv z; f \leftarrow v; T_b\ e_1; z$$

noting that $p_2 = \mathsf{id}$ when $f = f'$, from the definition of the function.

	Assertion	**Reasoning**
	$T_b\ (z; f \leftarrow v; (e_0 + a; q; (x : \tau)); z)$	
$\equiv$	$T_b\ z; f \leftarrow v; e_0; z + z; a; q; (x : \tau); z$	[Dist-R, Dist-L]
$\equiv$	$(T_b\ z; f \leftarrow v; e_0; z) + z; T_b\ a; q; (x : \tau); z$	[Definition of T]
$\equiv$	$(T_{\theta_3\ b}\ z; e_2; z) + z; T_b\ a; q; (x : \tau); z$	[IH]
$\equiv$	$z; (T_{\theta_3\ b}\ e_2); z + z; T_b\ a; q; (x : \tau); z$	[Definition of T]
$\equiv$	$z; (T_{\theta_3\ b}\ e_2); z + z; e_2; z$	[Lemma 66]
$\equiv$	$z; T_{\theta_3\ b}\ e_2; z + z; (T_{\theta_1\ b}\ p_1); z$	[Definition of θ_1]
$\equiv$	$z; ((T_{\theta_3\ b}\ e_2) + (T_{\theta_1\ b}\ p_1)); \mathsf{id}; z$	[Dist-R, Dist-L]
$\equiv$	$z; ((T_{\theta_3\ b}\ e_2) + (T_{(\theta_3 \circ \theta_1)\ b}\ p_1)); \mathsf{id}; z$	[Lemma 67]
$\equiv$	$z; ((T_{(\theta_3 \circ \theta_1)\ b}\ e_2) + (T_{(\theta_3 \circ \theta_1)\ b}\ p_1)); \mathsf{id}; z$	[Definition of θ_1]
$\equiv$	$T_{(\theta_3 \circ \theta_1)\ b}\ (z; (e_2 + p_1); \mathsf{id}; z)$	[Definition of T]

The remaining goals are immediate.

Subcase $f \neq f'$. After reducing $\mathsf{ext}_f\ s\ f' \leftarrow v\ e$ when $f \neq f'$, we must show the following.

$$T_b\ (z; f' \leftarrow v; (e_0 + a; q; (x : \tau)); z) \equiv T_{(\theta_3 \circ \theta_1)\ b}\ (z; (e_2 + p_1); f' \leftarrow v; z)$$

From the IH, we have

$$T_b\ (z; f' \leftarrow v; e_0; z) \equiv T_{\theta_3 b}\ (z; e_2; \mathsf{id}; z)$$

And from Lemma 66, we have

$$z; e_2; f' \leftarrow v; z \equiv z; f' \leftarrow v; T_b\ e_1; z$$

noting that $p_2 = f' \leftarrow v$ when $f \neq f'$, from the definition of the function.

	Assertion	**Reasoning**
	$T_b\ (z; f' \leftarrow v; (e_0 + a; q; (x : \tau)); z)$	
$\equiv$	$T_b\ z; f' \leftarrow v; e_0; z + z; f' \leftarrow v; a; q; (x : \tau); z$	[Dist-R, Dist-L]
$\equiv$	$(T_b\ z; f' \leftarrow v; e_0; z) + z; f' \leftarrow v; T_b\ a; q; (x : \tau); z$	[Definition of T]
$\equiv$	$(T_{\theta_3\ b}\ z; e_2; z) + z; f' \leftarrow v; T_b\ a; q; (x : \tau); z$	[IH]

$\equiv$	$z; (T_{\theta_3\ b}\ e_2); z + z; f' \leftarrow v; T_b\ a; q; (x : \tau); z$	[Definition of T]
$\equiv$	$z; (T_{\theta_3\ b}\ e_2); z + z; e_2; f' \leftarrow v; z$	[Lemma 66]
$\equiv$	$z; T_{\theta_3\ b}\ e_2; z + z; (T_{\theta_1\ b}\ p_1); f' \leftarrow v; z$	[Definition of θ_1]
$\equiv$	$z; ((T_{\theta_3\ b}\ e_2) + (T_{\theta_1\ b}\ p_1)); f' \leftarrow v; z$	[Dist-R, Dist-L]
$\equiv$	$z; ((T_{\theta_3\ b}\ e_2) + (T_{(\theta_3 \circ \theta_1)\ b}\ p_1)); f' \leftarrow v; z$	[Lemma 67]
$\equiv$	$z; ((T_{(\theta_3 \circ \theta_1)\ b}\ e_2) + (T_{(\theta_3 \circ \theta_1)\ b}\ p_1)); f' \leftarrow v; z$	[Definition of θ_1]
$\equiv$	$T_{(\theta_3 \circ \theta_1)\ b}\ (z; (e_2 + p_1); f' \leftarrow v; z)$	[Definition of T]

The remaining goals are immediate.

Case $r = (x : (\mathsf{R}, \mathsf{W}))$**.** After reducing, we have

$$
\begin{array}{l}
\mathsf{let}\ fs = s(x)\ \text{fresh fields}\ \mathsf{in} \\
\mathsf{let}\ \tau = \mathsf{typeof}\ e\ \mathsf{in} \\
\mathsf{let}\ \theta\ = \\
\quad (\lambda b, z. \\
\qquad \mathsf{let}\ e_1, p, \mathsf{id} = \mathsf{ext}_f\ s\ (T_b\ x)\ (T_b\ e)\ \mathsf{in} \\
\qquad \mathsf{if}\ z = w_1\ \mathsf{then}\ e_1 \\
\qquad \mathsf{else\ if}\ z = w_2\ \mathsf{then}\ p \\
\qquad \mathsf{else}\ T_b(z))\ \mathsf{in} \\
\mathsf{let}\ e' = (w_1 : (\emptyset, f \cup fs) \cup \tau)\ \mathsf{in} \\
\mathsf{let}\ p' = (w_2 : (\mathsf{R} \cup fs, \mathsf{W} \setminus f))\ \mathsf{in} \\
(e', p', \theta)
\end{array}
$$

From Lemma 66, we have that $\theta_1 = \mathsf{id}$, e_1 and p are closed, and $e_1; p \equiv (T_b\ (x : (\mathsf{R}, \mathsf{W})); e)$. We must show

$$T_b\ z; (x : (\mathsf{R}, \mathsf{W})); e; z \equiv T_{\theta\ b}\ z; e'; p'; z$$

	Assertion	**Reasoning**
	$T_b\ z; (x : (\mathsf{R}, \mathsf{W})); e; z$	
$\equiv$	$z; (T_b\ (x : (\mathsf{R}, \mathsf{W})); e); z$	[Definition of T]

$\equiv$	$z; e_1; p; z$	[Lemma 66]
$\equiv$	$T_{\theta\ b}\ z; e'; p'; z$	[Definition of T and θ]

The remaining goals are immediate.

□

C.2.4 If De-nesting

Lemma 69 (If flattening preserves semantics)**.** *For all well-typed predicates a and b and policies p and q and r, the following equivalence holds.*

$$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ \mathsf{else}\ r \equiv \mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else}\ (\mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ r)$$

Proof. The proof follows by a series of applications of the equational axioms.

	Assertion	**Reasoning**
	$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ \mathsf{else}\ r$	
$\equiv$	$a; (b; p + \neg b; q) + \neg a; r$	Desugar.
$\equiv$	$a; b; p + a; \neg b; q + \neg a; r$	Dist-L, Dist-R.
$\equiv$	$a; b; p + \neg b; a; q + \neg a; r$	BA-Seq-Comm.
$\equiv$	$a; b; p + \neg b; a; q + \neg a; \mathsf{id}; r$	KA-Seq-One.
$\equiv$	$a; b; p + \neg b; a; q + \neg a; (\mathsf{id} + \neg b); r$	BA-Plus-One.
$\equiv$	$a; b; p + \neg b; a; q + (\neg a + \neg b; \neg a); r$	Dist-L.
$\equiv$	$a; b; p + \neg b; a; q + \neg a; r + \neg b; \neg a; r$	Dist-R.
$\equiv$	$a; b; p + \neg b; a; q + \neg a; \neg a; r + \neg b; \neg a; r$	BA-Seq-Idem.
$\equiv$	$a; b; p + \mathsf{drop} + \neg b; a; q + \neg a; \neg a; r + \neg b; \neg a; r$	KA-Plus-Zero.
$\equiv$	$a; b; p + \neg a; a; q + \neg b; a; q + \neg a; \neg a; r + \neg b; \neg a; r$	PA-Contra, KA-Seq-Zero.
$\equiv$	$a; b; p + (\neg a + \neg b); a; q + (\neg a + \neg b); \neg a; r$	Dist-R.
$\equiv$	$a; b; p + \neg a; b; a; q + \neg a; b; \neg a; r$	De Morgan's (Lemma 55).
$\equiv$	$a; b; p + \neg a; b(a; q + \neg a; r)$	Dist-L, Dist-R.
$\equiv$	$\mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else}\ (\mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ r)$	Resugar.

□

Lemma 70 (If de-nesting preserves semantics)**.** *For all well-typed predicates a, policies p and q, and fields f, if f is a fresh field, then*

$$f \leftarrow 0; \mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q; f \leftarrow 0 \equiv f \leftarrow 0; (\mathsf{denest}\ \mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q); f \leftarrow 0$$

Proof.

	Assertion	**Reasoning**
	$f \leftarrow 0; (a; p + \neg a; q); f \leftarrow 0$	[Desugaring]
$\equiv$	$f \leftarrow 0; (a; p + \neg a; q); f \leftarrow 0$	
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 0$	
$+$	$f \leftarrow 0; \neg a; q; f \leftarrow 0$	[Dist-L, Dist-R]
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 0$	
$+$	$f \leftarrow 0; f = 0; \neg a; q; f \leftarrow 0$	[Mod-Match]
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 0$	
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; f = 0; q; f \leftarrow 0$	[Seq-One, freshness]
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 1; f \leftarrow 0$	
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; f = 0; q; f \leftarrow 0$	[Mod-Mod]
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 1; \neg(f = 0); \mathsf{id}; f \leftarrow 0$	[Lemma 56,
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; f = 0; q; f \leftarrow 0$	Mod-Match]
$\equiv$	drop	
$+$	$f \leftarrow 0; a; p; f \leftarrow 1; \neg(f = 0); \mathsf{id}; f \leftarrow 0$	
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; f = 0; q; f \leftarrow 0$	
$+$	drop	[Plus-Zero-Id]
$\equiv$	$f \leftarrow 0; a; p; f \leftarrow 1; f = 0; q; f \leftarrow 0$	
$+$	$f \leftarrow 0; a; p; f \leftarrow 1; \neg(f = 0); \mathsf{id}; f \leftarrow 0$	
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; f = 0; q; f \leftarrow 0$	[Mod-Match,
$+$	$f \leftarrow 0; \neg a; \mathsf{id}; \neg(f = 0); \mathsf{id}; f \leftarrow 0$	BA-Anihilate]
$\equiv$	$f \leftarrow 0; (a; p; f \leftarrow 1 + \neg a; \mathsf{id});$	
	$(f = 0; q + \neg f = 0; \mathsf{id}); f \leftarrow 0$	[Dist-R, Dist-L]
$\equiv$	$f \leftarrow 0; \mathsf{if}\ a\ \mathsf{then}\ p; f \leftarrow 1\ \mathsf{else}\ \mathsf{id};$	
	$\mathsf{if}\ f = 0\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{id}; f \leftarrow 0$	[Sugaring]

□

Lemma 71. *For all well-typed predicates a and b and policies p and q, if $a; p \equiv p; a$, then*

$$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{id}); q\ \mathsf{else}\ \mathsf{id} \equiv (\mathsf{if}\ a; b\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{id}); \mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{id}.$$

Proof.

	Assertion	Reasoning
	$\mathsf{if}\ a\ \mathsf{then}\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{id}); q\ \mathsf{else}\ \mathsf{id}$	
$\equiv$	$a; (b; p + \neg b); q + \neg a$	Desugar.
$\equiv$	$a; b; p; q + a; \neg b; q + \neg a$	Dist-L, Dist-R.
$\equiv$	$a; b; p; q + \neg b; a; q + \neg a$	BA-Seq-Comm.
$\equiv$	$a; b; p; q + \neg b; a; q + (\mathsf{id} + \neg b); \neg a$	BA-Plus-One, KA-Zero-Seq.
$\equiv$	$a; b; p; q + \neg b; a; q + \neg a + \neg b; \neg a$	Dist-L.
$\equiv$	$a; b; p; q + (\neg a + \neg b); a; q + (\neg a + \neg b); \neg a$	Dist-R, PA-Contra, KA-Seq-Zero, KA-Plus-Zero.
$\equiv$	$a; b; p; q + \neg a; b; a; q + \neg a; b; \neg a$	De Morgan's (Lemma 55).
$\equiv$	$a; b; p; q + \mathsf{drop} + \neg a; b; a; q + \neg a; b; \neg a$	KA-Plus-Zero.
$\equiv$	$a; a; b; p; q + a; \neg a; b; p + \neg a; b; a; q + \neg a; b; \neg a$	KA-Zero-Seq, BA-Seq-Idem, PA-Contra.
$\equiv$	$a; b; p; a; q + a; b; p; \neg a + \neg a; b; a; q + \neg a; b; \neg a$	Hypothesis, KAT-Commute.
$\equiv$	$(a; b; p + \neg a; b); (a; q + \neg a)$	Dist-L, Dist-R.
$\equiv$	$(\mathsf{if}\ a; b\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{id}); \mathsf{if}\ a\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{id}$	Resugar.

□

C.2.5 Table Replacement

Ultimately, all the logic in a user's configuration policy must be embedded into tables. In this step, we transform each non-table element into a (logical) table, coupled with a table binding that installs the original element to the new table at run time.

Definition 24. *Let y be a fresh table name.* $\mathsf{pack}\ p = (y : (\mathsf{typeof}\ p), (\lambda\ b, x.\ \mathsf{if}\ x = y\ \mathsf{then}\ T_b\ p\ \mathsf{else}\ T_b\ x)$.

Tables begin their lives empty, waiting to be populated with packet processing rules. The $\mathsf{pack}\ p : \tau$ function replaces p with a fresh table y and produces a binding transformer that populates y with p.

Lemma 72 (Table extension)**.** *For all policies p, table names y, and table bindings b, let $q, \theta = \mathsf{pack}\ p$.*

$$T_b\ p \equiv T_{\theta b}\ q$$

Proof. After unrolling $\mathsf{pack}\ p$, we have that $T_{\theta b}\ q = T_b\ p$.

□

Definition 25 (Tableify)**.**

$$\begin{array}{lcl}
\mathsf{tableify}\ a & = & \mathsf{pack}\ a \\
\mathsf{tableify}\ f = v & = & \mathsf{pack}\ f = v \\
\mathsf{tableify}\ f \leftarrow v & = & \mathsf{pack}\ f \leftarrow v \\
\mathsf{tableify\ if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q & = & \mathsf{pack\ if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q \\
\mathsf{tableify}\ p + q & = & \mathsf{let}\ x_1, \theta_1 = \mathsf{pack}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ x_2, \theta_2 = \mathsf{pack}\ q\ \mathsf{in} \\
 & & (x_1 + x_2, \theta_2 \circ \theta_1) \\
\mathsf{tableify}\ p; q & = & \mathsf{let}\ x_1, \theta_1 = \mathsf{pack}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ x_2, \theta_2 = \mathsf{pack}\ q\ \mathsf{in} \\
 & & (x_1; x_2, \theta_2 \circ \theta_1) \\
\mathsf{tableify}\ p \,||\, q & = & \mathsf{let}\ x_1, \theta_1 = \mathsf{pack}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ x_2, \theta_2 = \mathsf{pack}\ q\ \mathsf{in} \\
 & & (x_1 \,||\, x_2, \theta_2 \circ \theta_1)
\end{array}$$

The $\mathsf{tableify}\ p$ function is a simple morphism that replaces each atomic element in p with a new table and constructs a binding transformation that inserts the original atomic element into its table. Additionally, if statements are also replaced with tables.

Lemma 73. *For all policies p and q, table bindings b, and binding transformations θ, if* $\mathsf{tableify}\ p = (q, \theta)$ *then $T_b\ p \equiv T_{\theta\ b}\ q$.*

Proof. By induction on p. Atomic cases follow from Lemma 72, and combinator cases follow from the induction hypothesis.

□

C.2.6 Dynamic Programming

So far, we have pipelined multicast, de-nested if statements, and table-fied every other atomic element, leaving us with a policy consisting of a list of tables followed by a sum of multicast tables followed by a list of tables, where each list is joined by either sequential or concurrent composition. Below, we develop an algorithm for placing policies joined only by sequential composition. In order to find the best ordering, we simply try all combinations of sequentializations of the concurrent compositions.

Now, suppose we have the following restricted language:

$$\begin{array}{rcl} p, q & ::= & \mathsf{table}[n] : \tau \\ & | & p; q \\ & | & p \,||\, q \end{array}$$

And we have a sequence of tables $t_1; t_2; \ldots; t_n$.

The RMT pipeline. Recall that in the architecture of the RMT chip, each match stage has sixteen blocks of 40b by 2048 entry TCAMs and 80b by 1024 entry SRAMs. For now, we'll focus on TCAM memory.

Definition 26 (TCAM Block Cost)**.**

$$\mathsf{blocks}\ a \;=\; \lceil(\mathsf{width}\ a)/40\rceil * \lceil(\mathsf{height}\ a)/2048\rceil$$

Given the policy $p; q$, there are two options for deploying it to a multi-stage switch. The first is to place p and q in separate tables, such that q appears after p. The second is to compile p and q into a single big table at the controller, and then install this

table on as many physical tables as necessary. The cost of the former is $\mathcal{O}(p+q)$ and the latter is $\mathcal{O}(pq)$.

Definition 27 (Compilation)**.**

$$
\begin{array}{lcl}
\mathit{compile}\ \mathsf{table}[n] : \tau & = & \mathsf{table}[n] : \tau \\
\mathit{compile}\ p; q & = & \mathsf{let}\ \mathsf{table}[n_p] : (\rho_p, \alpha_p) = \mathit{compile}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ \mathsf{table}[n_q] : (\rho_q, \alpha_q) = \mathit{compile}\ q\ \mathsf{in} \\
 & & \mathsf{table}[n_p * n_q] : (\rho_p \cup \rho_q, \alpha_p \cup \alpha_q) \\
\mathit{compile}\ p \,||\, q & = & \mathsf{let}\ \mathsf{table}[n_p] : (\rho_p, \alpha_p) = \mathit{compile}\ p\ \mathsf{in} \\
 & & \mathsf{let}\ \mathsf{table}[n_q] : (\rho_q, \alpha_q) = \mathit{compile}\ q\ \mathsf{in} \\
 & & \mathsf{table}[n_p + n_q] : (\rho_p \cup \rho_q, \alpha_p \cup \alpha_q)
\end{array}
$$

This calculation of table size is, in many cases, conservative. For example, suppose p and q are tables that both match on srcip. Their sequential compilation will, at worst, be additive, not multiplicative. However, suppose p matches srcip and dstip, whereas q matches dstip and dstport. This is similar to a join in a relational model, and so we may potentially take the cross product of fields.

Even worse, suppose p and q both match on srcip and dstip. If every rule in each table matched on each field, we could combine the tables at a fixed cost. However, suppose p opted to only match on srcip and p on dstip, underutilizing their permissions. Now compilation produces the cross product.

Definition 28 (Table Cost)**.**

$$
\begin{array}{lcl}
\mathit{tables}\ a & = & \lceil (\mathsf{blocks}\ a)/16 \rceil \\
\mathit{tables}\ p; q & = & \min\{\mathit{tables}\ p + \mathit{tables}\ q,\ \mathit{tables}\ (\mathit{compile}\ p; q)\} \\
\mathit{tables}\ p \,||\, q & = & \min\{\mathit{tables}\ p + \mathit{tables}\ q,\ \mathit{tables}\ (\mathit{compile}\ p \,||\, q)\}
\end{array}
$$

The problem. Given a sequence of of atoms $a_1, \ldots, a_n$, determine the smallest sequence of tables required to contain the sequence of atoms.

The algorithm. For convenience, let a_{ij} represent an in-order numbering of the leaves of the abstract syntax tree, starting with a_1 as the leftmost leaf. For example, $a_1; (a_2 \mid\mid a_3); a_4$, and $a_{23} = a_2 \mid\mid a_3$. When such a subset spans the boundary of a concurrent composition, sequentialize it. For example, $a_{12} = a_1; a_2$, and $a_{34} = a_3; a_4$.

```
   input : A sequence of a_{1n}
 1 let m[1...n, 1...n] and s[1...n-1, 2...n] be new tables;
 2 for i = 1 to n do
 3 |  m[i,i] = ⌈(blocks a_i)/16⌉;
 4 end
 5 for l = 2 to n do
 6 |  for i = 1 to n - l + 1 do
 7 |  |  j = i + l - 1;
 8 |  |  m[i,j] = ∞;
 9 |  |  for k = i to j - 1 do
10 |  |  |  q = min(m[i,k] + m[k+1,j], ⌈blocks compile a_{ij}/16⌉);
11 |  |  |  if q < m[i,j] then
12 |  |  |  |  m[i,j] = q;
13 |  |  |  |  s[i,j] = k;
14 |  |  |  end
15 |  |  end
16 |  end
17 end
18 return m and s
```

C.2.7 Combining Multicast Consolidation and Field Extraction

Lemma 74 (A summation can be split and connected by unique tags)**.** *For all summations* $\sum_i p_i; q_i; q'_i; p'_i$ *and pairs of policies* q_i, q'_i*, if (H1)* $q_i; q_j \equiv \mathsf{drop}$ *for all* $i \neq j$*, then* $(\sum_i p_i; q_i; q'_i; p'_i) \equiv (\sum_i p_i; q_i); (\sum_j q'_j; p'_j)$.

Proof. In essence, the proof reasons in reverse by observing that the summands of the cross product of the two summations are equivalent to drop when $i \neq j$.

	Assertion	**Reasoning**
	$(\sum_i p_i; q_i); (\sum_j q'_j; p'_j)$	
$\equiv$	$(\sum_{ij} p_i; q_i; q'_j; p'_j)$	Dist-L, Dist-R.
$\equiv$	$(\sum_i p_i; q_i; q'_i; p'_i)$	H1, KA-Seq-Zero, KA-Plus-Zero.

□

Lemma 75 (Summations with unique predicates in each summand are equivalent to an if)**.** *For all summations* $\sum_i a_i; p_i$*, if* $a_i; a_j \equiv \mathsf{drop}$ *for* $i \neq j$*, then*

$$\begin{aligned} \textstyle\sum_i a_i; p_i \quad \equiv \quad & \mathsf{if}\ a_1\ \mathsf{then}\ p_1\ \mathsf{else} \\ & \mathsf{if}\ a_2\ \mathsf{then}\ p_2\ \mathsf{else} \\ & \ldots \\ & \mathsf{drop} \end{aligned}$$

Proof. The proof goes by induction on the summation. The base case, where the summation is empty and thus equivalent to drop, is immediate, leaving us the case of $a; p + (\sum_j a_j; p_j)$. By KA-One-Seq and BA-Excl-Mid, we have the following.

$$a; p + (\sum_j a_j; p_j) \equiv a; p + (a + \neg a); (\sum_j a_j; p_j)$$

And we can distribute the summation using Dist-R.

$$\equiv a; p + a; (\sum_j a_j; p_j) + \neg a; (\sum_j a_j; p_j)$$

But, by our hypothesis, $a; a_j \equiv \mathsf{drop}$. Hence, following KA-Seq-Zero and KA-Plus-Zero, the middle summation falls away.

$$\equiv a; p + \neg a; (\sum_j a_j; p_j)$$

Applying the IH gives us an if statement, call it q, such that $(\sum_j a_j; p_j) \equiv q$. Substituting equals for equals and applying the if statement sugaring yields our goal.

$$\equiv \mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q$$

□

Bibliography

[1] Openflow forwarding abstractions working group charter, Apr 2013. See http://goo.gl/TtLtw0.

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

[3] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

[4] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[5] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming protocol-independent packet processors. See http://arxiv.org/abs/1312.1719, Dec 2013.

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, pages 99–110, 2013.

[7] Broadcom BCM56846 StrataXGS 10/40 GbE switch. See http://www.broadcom.com/products/features/BCM56846.php, 2014.

[8] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J., 1994.

[9] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[10] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[12] David K Gifford, Pierre Jouvelot, Mark A Sheldon, and James W O'Toole. Report on the fx-91 programming language. Technical report, DTIC Document, 1992.

[13] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, June 2013.

[14] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct 1969.

[16] C. A. R. Hoare, Bernhard M oller, Georg Struth, and Ian Wehrman. Concurrent kleene algebra. In *CONCUR*, pages 399–414, 2009.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[18] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, May 2015.

[19] Cliff B Jones. *Development methods for computer programs including a notion of interference*. Oxford University Computing Laboratory, 1981.

[20] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, May 2015.

[21] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[22] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, New York, NY, USA, 2014. ACM.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[24] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *I&C*, 110(2):366–390, May 1994.

[25] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *TACAS*, pages 14–33, Passau, Germany, March 1996.

[26] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[27] Dexter Kozen. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Computer Science Department, Cornell University, November 2003.

[28] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In *CL*, Jul 2000.

[29] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 12–23, New York, NY, USA, 2003. ACM.

[30] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 147–162, New York, NY, USA, 2012. ACM.

[31] Alex X. Liu, Chad R. Meiners, and Eric Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 18(2):490–500, Apr 2010.

[32] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.

[33] James McCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. Extending SDN to large-scale networks. In *ONS*, 2013.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computing Communications Review*, 38(2):69–74, 2008.

[35] Chad R. Meiners, Alex X. Liu, Eric Torng, and Jignesh Patel. Split: Optimizing space, power, and throughput for tcam-based classification. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and*

Communications Systems, ANCS '11, pages 200–210, Washington, DC, USA, 2011. IEEE Computer Society.

[36] B. Möller. Calculating with pointer structures. In *Algorithmic Languages and Calculi. Proc. IFIP TC2/WG2.1 Working Conference*, February 1997.

[37] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, Jan 2012.

[38] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, Apr 2013.

[39] Mike Neil. Root cause analysis for recent windows azure service interruption in western europe. http://azure.microsoft.com/blog/2012/08/02/root-cause-analysis-for-recent-windows-azure-service-interruption-in-western-europe/, 2012.

[40] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[41] Openflow switch specification, version 1.0.0. See https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf, 2009.

[42] Recep Ozdag. Intel Ethernet Switch FM6000 Series - software defined networking. See goo.gl/AnvOvX, 2012.

[43] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[44] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[45] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 11–24, New York, NY, USA, 2014. ACM.

[46] Scott Shenker. The future of networking, and the past of protocols. The Open Networking Summit, 2011.

[47] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *PODS*, pages 237–249, 1987.

[48] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[49] The AWS Team. Summary of the amazon ec2 and amazon rds service disruption in the us east region. https://aws.amazon.com/message/65648/, 2011.

[50] M Tofte and J P Talpin. Implementation of the typed lambda calculus using a stack of regions. In *POPL*, January 1994.

[51] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th Annual Workshop on Microprogramming*, MICRO 15, pages 125–133, Piscataway, NJ, USA, 1982. IEEE Press.

[52] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[53] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *HotSDN*, pages 43–48, 2012.

[54] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[55] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, pages 137–146, New York, NY, USA, 1990. ACM.

[56] Minlan Yu, Jennifer Rexford, Xin Sun, Sanjay G. Rao, and Nick Feamster. A survey of virtual LAN usage in campus networks. *IEEE Communications Magazine*, 49(7):98–103, 2011.