Concurrent NetCore: From Policies to Pipelines

Cole Schlesinger
Princeton University

Michael Greenberg
Princeton University

David Walker
Princeton University

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 [5] and FlexPipe [14] 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 [4]. 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.

We present 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 and we give a multi-pass compilation algorithm that translates graphs and routing policies to the RMT model.

1. Introduction

Over the past several years, a new networking technology known as *Software-Defined Networking* (SDN) has emerged as a viable competitor to traditional networking infrastructure. In a software-defined network, 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 each switch in the hope of inducing behavior that conforms to a global (and often poorly specified) network policy.

SDN has had a tremendous impact in the networking community, both for industry and academia. Google has adoped SDN to manage its internal backbone, which transmits all its intra-datacenter traffic—making it one of the largest networks in the world [9], 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 [10]—includes the owners of most of the largest networks in the world (Google, Facebook, Microsoft, etc) and its 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 Software-Defined Networks 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.

OpenFlow 1.0: successes and failures. The OpenFlow protocol is a popular protocol for communication between the controller and switches. The first version, OpenFlow 1.0 [10], supported a simple abstraction: 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 [7], Procera [15], Maple [16], FlowLog [13] 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 (eg. 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 [6].

In addition to building fixed-pipeline ASICs, switch hardware vendors are also developing more programmable hardware pipelines. For example, the RMT design [5] 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 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 a 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 [14] 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 [4] (See Figure 1). In this proposal, switch configuration is divided into two phases: table configuration and table population.

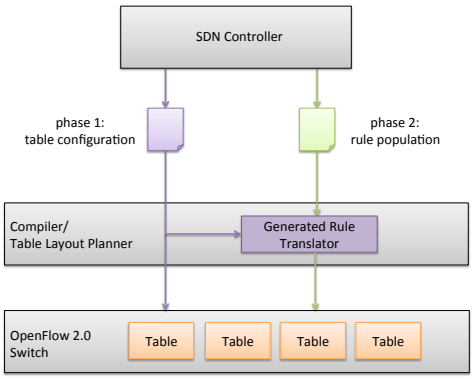

Figure 1. Architecture of an OpenFlow 2.0 System

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 necessary for executing the rules within a table.

We call the tables communicated from controller to switch *abstract*, because they do not necessarily correspond directly to the *concrete* 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. 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 Section 2.

Contributions of this paper. The central contribution of this paper is the design of a new language for programming OpenFlow 2.0 switches. This compiler 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 [7, 11] and NetKAT [3].[1] Like NetCore and NetKAT, Concurrent NetCore consists of a small number of primitive operations for specifying packet processing, plus combinators for constructing more com-

[1] Because we focus on programming individual switches in this paper, 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*.

plex packet processors from simpler ones. Concurrent NetCore introduces the following new 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 provides a 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 of the paper 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 definition and proof of soundness of the type system 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 [5]. In doing so, we show how to produce *policy transformation functions* that will map abstract policy updates in to 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.

A technical appendix is available that includes a full presentation of the compilation algorithm, theorems, and proofs [1].

The following section introduces CNC in greater detail through a series of examples, while Section 3 presents a formal semantics for CNC, and Section 4 describes its metatheory. The models of both the RMT and Intel FlexPipe architectures are described in Section 5, followed by our compilation algorithm in Section 6. Section 7 describes related work, and we conclude in 8.

2. CNC by example

In this section, 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. Because CNC can model both ends of the spectrum, it can serve as a common intermediate within an OpenFlow 2.0 compilation system. Section 6 will illustrate this idea via algorithms that demonstrate how to transform our high-level user policies in to components for placement in RMT tables.

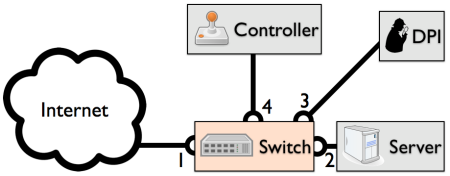

Figure 2. A simple network.

2.1 Simple switch policies

Consider the picture in Figure 2. This picture presents several devices, a switch, a controller, a server and a DPI 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 to specify how it forwards packets in and out of its ports.

In general, we model packets as records with a number of fields. Our examples typically use an idealized collection of fields such as src (the packet's source IP address), in (the port the packet arrives on), and out (the port a packet should leave on).Switch policies are functions that map packets to sets of packets. For example, a policy that drops all packets will map any packet to the empty set of packets. A policy that forwards packets from the internet (port 1) to the server (port 2) will map packets with in field 1 to a packet with out 2. A policy that forwards packets from the internet to both the DPI box and the server will map packets with $\mathsf{in} = 1$ to a pair of packets with $\mathsf{out} = 2$ and $\mathsf{out} = 3$.

We build our policies out of a collection of primitive operations and policy combinators. The simplest primitive filters packets based on the contents of a single field. For example, when applied to a packet pk, the test $\mathsf{src} = 10.0.0.1$ returns $\{pk\}$ when pk's src field is 10.0.0.1 and returns the empty set of packets otherwise. Using such tests as well as standard boolean connectives *and* (;), *or* (+) and *not* ($\neg$), one can easily build up a function on packets that implements a firewall (either dropping each packet or returning it unchanged). For example, we might want to implement the following firewall w on the switch in Figure 2. It admits ssh or $http$ traffic on port 1, but blocks all other traffic arriving on port 1. All traffic on ports other than 1 is allowed.

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

In order to make changes to packets, we use the assignment primitive $f \leftarrow value$. Complex policies may perform the actions of a set of simpler policies in series using the sequential composition operator $(p_1; p_2)$. Alternatively, a policy may copy a packet and perform both p_1 and p_2 on the separate copies, taking the union of their results $(p_1 + p_2)$. We have reused the symbols ";" and "+" (conjunction and disjunction) here as it turns out their semantics as logical predicates coincides with their semantics as policy combinators (the boolean algebra is a sub-algebra of the policy algebra).

As an example, to define a static routing policy r for our switch, we might write the following policy.

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

The policy above has the effect of routing packets from port 1 to port 2 and from port 2 to port 1. In more detail, it first copies the incoming packet (+). Then, in the first branch, it tests whether the input port is 1; if not, the packet is dropped; if so, the out field is

assigned 2. The second branch is dual, forwarding packets from port 2 to port 1. The guards on each branch guarantee that the 'copying' is purely notional; in general, one codes the conditional statement $\mathsf{if}\ a\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$ as $a; p_1 + \neg a; p_2$.

The above features are not new—they are present in NetCore [11, 12] and NetKAT [3]. However, in order to serve as a configuration language for OpenFlow 2.0, we require a couple of additional features, as well as the development of 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). We model rule population as a set of *table bindings* b, i.e., a closing substitution.

A second key extension is concurrency, written $p_1 \mid\mid p_2$. 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).

$$\begin{array}{rcl} m & = & (x : (\{\mathsf{typ}, \mathsf{src}\}, \{\mathsf{out}\})) \\ p & = & w \mid\mid (r + m) \end{array}$$

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, 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.

2.2 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 [5] and FlexPipe [14] architectures. Both chips offer differently architectured fixed pipelines connecting reconfigurable tables.

RMT. 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.

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

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. Overall, the RMT pipeline consists of the ingress pipeline, followed by the multicast stage, followed by the egress pipeline.

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

FlexPipe. 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.

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

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.

$$\begin{array}{rcl} \mathsf{multicast} & = & \mathsf{mirror}; \mathsf{egress}; \mathsf{flood} \\ \mathsf{pipeline} & = & \mathsf{diamond}; \mathsf{multicast} \end{array}$$

We present models of both RMT and FlexPipe (including mirror, egress and flood) in greater detail in Section 5.

3. Concurrent NetCore

We define the syntax of Concurrent NetCore in Figure 3. 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 [3]. 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. 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.

Fields	$f \in \mathsf{F} ::=$	$f_1 \mid \cdots \mid f_k$	
Packets	$pk \in \mathsf{PK} ::=$	$\mathsf{F} \rightharpoonup \mathsf{Val}$	
Variables	x, y		
Types	$\tau \in$	$\mathcal{P}(\mathsf{R}) \times \mathcal{P}(\mathsf{W})$	
Predicates	$a, b ::=$	id	*Identity (True)*
		$\mid \mathsf{drop}$	*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	$\sigma ::=$	$\langle p, \delta \rangle$	
Packet trees	$\delta ::=$	$\langle \mathsf{PK}, \mathsf{W} \rangle$	*Leaves*
		$\mid \langle \mathsf{par}\ \delta_1\ \delta_2 \rangle$	*Parallel processing*
		$\mid \langle \mathsf{not}_{\mathsf{PK}}\ \delta \rangle$	*Pending negation*
		$\mid \langle \mathsf{con}_{\mathsf{W}}\ \delta_1\ \delta_2 \rangle$	*Concurrent processing*

Figure 3. Packets, types, and predicate/policy syntax

As explained in Section 2, 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.

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.

3.1 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.

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 4, we define a type system that guarantees the strong normalization of all concurrent executions, 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.

Using PK to range over sets of packets, we define the states σ for the small-step operational semantics $\sigma \rightarrow \sigma'$ in Figure 3. 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: which packets, or packet components, are the different branches of the parallel and concurrent compositions working on? When processing a negation, from what set of packets will we take the complement?

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 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* satsfying a. 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 current write permission, so we can restore it when p and q are done processing.

We write $\sigma \rightarrow \sigma'$ 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 to hand, we define the step relation in Figure 4. The following invariants of evaluation and well-typed policies may be of use while reading through Figure 4 the following.

- 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.
- 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 run p to completion (SEQL), and then we run q on the resulting packets (SEQR). A special packet tree branch is not necessary, because q runs on any and all output that p produces. 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.

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,

Packet operations

$$pk[f := v] = \lambda f' \,.\, \begin{cases} v & f = f' \\ pk(f') & \text{otherwise} \end{cases} \qquad pk \setminus \mathsf{F} = \lambda f \,.\, \begin{cases} \bot & f \in \mathsf{F} \\ pk(f) & \text{otherwise} \end{cases} \qquad \mathsf{PK} \setminus \mathsf{F} = \{pk \setminus \mathsf{F} \mid pk \in \mathsf{PK}\}$$

$$pk_1 \times pk_2 = \lambda f \,.\, \begin{cases} pk_1(f) & \text{when } f \notin \mathsf{Dom}\,(pk_2) \\ pk_2(f) & \text{when } f \notin \mathsf{Dom}\,(pk_1) \\ pk_1(f) & \text{when } pk_1(f) = pk_2(f) \end{cases} \qquad \mathsf{PK}_1 \times \mathsf{PK}_2 = \{\, pk_1 \times pk_2 | \, pk_1 \in \mathsf{PK}_1, pk_2 \in \mathsf{PK}_2\}$$

Reduction relation

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

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

$$\frac{f \in \mathsf{W}}{\langle f \leftarrow v, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \to \langle \mathsf{id}, \langle \{pk[f := v] \mid pk \in \mathsf{PK}\}, \mathsf{W}\rangle\rangle}\textsc{Modify} \qquad \frac{\langle p, \delta\rangle \to \langle p', \delta'\rangle}{\langle p; q, \delta\rangle \to \langle p'; q, \delta'\rangle}\textsc{SeqL} \qquad \frac{}{\langle \mathsf{id}; q, \delta\rangle \to \langle q, \delta\rangle}\textsc{SeqR}$$

$$\frac{}{\langle p + q, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \to \langle p + q, \langle \mathsf{par}\ \langle \mathsf{PK}, \mathsf{W}\rangle\ \langle \mathsf{PK}, \mathsf{W}\rangle\rangle\rangle}\textsc{ParEnter} \qquad \frac{\langle p, \delta_p\rangle \to \langle p', \delta'_p\rangle}{\langle p + q, \langle \mathsf{par}\ \delta_p\ \delta_q\rangle\rangle \to \langle p' + q, \langle \mathsf{par}\ \delta'_p\ \delta_q\rangle\rangle}\textsc{ParL}$$

$$\frac{\langle q, \delta_q\rangle \to \langle q', \delta'_q\rangle}{\langle p + q, \langle \mathsf{par}\ \delta_p\ \delta_q\rangle\rangle \to \langle p + q', \langle \mathsf{par}\ \delta_p\ \delta'_q\rangle\rangle}\textsc{ParR} \qquad \frac{}{\langle \mathsf{id} + \mathsf{id}, \langle \mathsf{par}\ \langle \mathsf{PK}_p, \mathsf{W}\rangle\ \langle \mathsf{PK}_q, \mathsf{W}\rangle\rangle\rangle \to \langle \mathsf{id}, \langle \mathsf{PK}_p \cup \mathsf{PK}_q, \mathsf{W}\rangle\rangle}\textsc{ParExit}$$

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

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

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

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

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

Figure 4. Concurrent NetCore operational semantics

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 5, Section 4). 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 4 is the ultimate argument, 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 $p \;{}_{\{f_1\}}||_{\{f_2\}}\; q$ on PK will yield

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

  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!

4. Metatheory

The operational semantics of Section 3.1/Figure 4 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 $f \notin \mathsf{Dom}\,(pk)$.
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 applying (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 5, 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 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. Tables 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).

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash p : \tau_1 \qquad \Gamma \vdash q : \tau_2}{\Gamma \vdash p; q : (\tau_1 \cup \tau_2)}\textsc{Seq}$$

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

Figure 5. Concurrent NetCore typing rules

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 4 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 adhere, then we 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. When proving

normalization, we add a condition (2) showing that fields not in the write permission W are "read-only". We need this condition in the concurrency case of the proof to show that the cross-product in (ConExit) is well defined.

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

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

then $\langle p, \langle \mathsf{PK}, \mathsf{W} \rangle\rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W} \rangle\rangle$ *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. □

Next we show that our calculus is confluent—even for ill typed terms. This is 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* $\sigma \rightarrow^* \sigma_1$ *and* $\sigma \rightarrow^* \sigma_2$ *then there exists* σ' *such that* $\sigma_1 \rightarrow^* \sigma'$ *and* $\sigma_2 \rightarrow^* \sigma'$.

Proof. By induction on the derivation of $\sigma \rightarrow^* \sigma_1$, proving the single-step diamond property first. □

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 [3], though we (a) do away with histories, and (b) make the quantification in the definition of sequencing explicit. Since our policies are 'switch-local', we omit Kleene star.

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

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

Proof. By induction on $\cdot \vdash p : \tau$. □

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:

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

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

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

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* $\langle p \ \mathsf{w}_p||\mathsf{w}_q\ q, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W}\rangle\rangle$ *iff* $\langle p; q, \langle \mathsf{PK}, \mathsf{W}\rangle\rangle \rightarrow^* \langle \mathsf{id}, \langle \mathsf{PK}', \mathsf{W}\rangle\rangle$.

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 $PK \setminus \mathsf{W}_q$ and on PK (when $\mathsf{R}_p \cap \mathsf{W}_q = \mathsf{W}_p \cap W_q = \emptyset$). □

5. Modeling RMT and FlexPipe architectures

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 6 presents models for the RMT and FlexPipe architectures. Both the RMT and FlexPipe 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 physical tables be of the form above.

We model run-time updates using a partial function b that maps table variables to closed policies. 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 at run time.

If statements. Before examining each physical table stage in detail, 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

Run-time updates

$$
\begin{array}{rcl}
b & \in & \mathsf{TVar} \rightharpoonup \mathsf{Policy} \\
T & \in & (\mathsf{TVar} \rightharpoonup \mathsf{Policy}) \to \mathsf{Policy} \to \mathsf{Policy}
\end{array}
$$

Physical tables

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

RMT model

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

FlexPipe model

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

Figure 6. Modeling RMT and Intel FlexPipe.

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 $(\mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ q)$ for $(a; p + \neg a; q)$.

Physical tables. Each physical table $b(x : \tau)$ 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. These fields implicitly determine the width of the match. The metadata field act_i holds an action identifier A_{ij}, a stand-in for the slightly more structured action languages of the RMT and FlexPipe chips. By construction, action selection is written to a metadata field unique to that match, allowing for the match stage to execute multiple matches in parallel. For example, $A_{11} \ldots A_{1k}$ might correspond to updating the src field, $A_{21} \ldots A_{2j}$ the dst field, and so on. 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 $\mathsf{do}_{A_{ij}}$, one for each A_{ij}. Each field $\mathsf{do}_{A_{ij}}$ 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.

A physical stage will only be well typed—and well behaved on execution—if the A_{ij} fields it has at its disposal don't overlap. That is, if A_{ij} and A_{ik} have overlapping effects—say A_{ij} sets the destination IP to one value while A_{ik} sets it to another—then the actions part of a physical stage $\mathsf{physical}_k$ would have a race condition. A key feature of our system is to reject such programs.

As an example, suppose we wanted to compile the routing policy (r) from Figure 2 as a physical table. First, let's fix two concrete action values—we'll say that a value of 1 means "modify the out field to 1" and a value of 2 means "modify the out field to 2." Our match stage is a single match predicated on the out field. The crossbar connects our action values (1 and 2) to the appropriate actions in the actions stage.

$$
\begin{array}{rcl}
\mathsf{match} & = & \mathsf{if\ in} = 1\ \mathsf{then\ act} \leftarrow 2 \\
& & \mathsf{else\ if\ in} = 2\ \mathsf{then\ act} \leftarrow 1 \\
& & \mathsf{else\ drop} \\
\\
\mathsf{crossbar} & = & \mathsf{if\ act} = 1\ \mathsf{then\ do}_1 \leftarrow 1 \\
& & \mathsf{else\ if\ act} = 2\ \mathsf{then\ do}_2 \leftarrow 1 \\
& & \mathsf{else\ drop} \\
\mathsf{actions} & = & \mathsf{if\ do}_1 = 1\ \mathsf{then\ out} \leftarrow 1 \\
& & \mathsf{else\ if\ do}_2 = 1\ \mathsf{then\ out} \leftarrow 2 \\
& & \mathsf{else\ id}
\end{array}
$$

Separating tables into three stages may seem excessive, but suppose our example also set a VLAN tag based on the out field. By splitting reading and writing into separate phases, the match stage for determining which VLAN to write can run concurrently with the match determining the output port. 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. 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 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 tables. 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 2.

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.

6. Compilation

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.

- **Multicast consolidation** transforms a policy with arbitrary occurrences of multicast ($+$) into a pipelined policy wherein multicast occurs at just a single stage.
- **Field extraction** moves modifications of a given field to an earlier stage of a pipelined policy.
- **Table fitting** partitions a pipelined policy into a sequence of tables, possibly combining multiple policy fragments into a single table.

Each pass takes a policy as input and produces an equivalent, refactored policy as well as a *binding transformer* as output. These binding transformers serve the same role as the "generated rule translator" from Figure 1. In other words, during the switch "population" phase, the controller will issue table bindings b—essentially, closing substitutions—in terms of the original policy pre-compilation. It is the binding transformer θ's job to transform these table bindings so that they can be applied sensibly to the post-compilation pipeline configured on the switch.

Definition 1 (Binding transformer)**.** *A binding transformer θ is an operator on table bindings b.*

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

As a brief example, let's look at how the first phase of compilation will work on the $r + m$ fragment of the example policy from Section 2. The first step is to apply multicast consolidation, using a function we call pipeline. But recall that m contains a table variable—which may introduce more multicast later. 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. Then, we can replace $r + m$ with the following:

$$\begin{array}{l}\mathsf{pipeline}\ s\ (r+m) = \\ \quad (f \leftarrow 0 + y : (\{\mathsf{typ}, \mathsf{src}\}, fs); f \leftarrow 1); \\ \quad (\mathsf{if}\ f = 0\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}); \\ \quad (\mathsf{if}\ f = 1\ \mathsf{then}\ z : (\{\mathsf{typ}, \mathsf{src}\} \cup fs, \{\mathsf{out}\})\ \mathsf{else}\ \mathsf{id})\end{array}$$

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.

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

Suppose an update arrives in the form of a 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' is always the identity function.

6.1 Multicast consolidation

There are two important differences between the kind of multicast that Concurrent NetCore offers and the kind supported by the RMT pipeline. 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.* The programming model showcased in our example is very different: first copy the packet, then process each copy to determine its fate. We use *multicast consolidation* to rewrite a high-level policy into a form with a distinct multicast stage. The next section describes how we use *field extraction* to extract potential modifications to a given field from a subpolicy—which we will use to isolate writes to the output port to the multicast stage.

Multicast consolidation works by replacing multicast with metadata indicating the "intent to multicast"—which is appropriately acted upon in the resulting consolidated multicast stage. To capture this, we define a syntactically restricted form that models consolidated packet duplication and tagging. This form is similar to the multicast stage presented in Figure 6 but slightly higher-level, in that it may contain table variables and additional field modifications—later compilation phases will factor these out.

Definition 2 (Consolidation stage)**.**

$$\begin{array}{lrcl}\textit{consolidation sequence} & m & ::= & \Pi_i f_i \leftarrow 1 \mid (x : \tau); \Pi_i f_i \leftarrow 1 \\ \textit{consolidation stage} & ms & ::= & \sum_i a_i; m_i \\ \textit{egress stage} & n & ::= & x : \tau \mid n; r \\ & & \mid & n; \mathsf{if}\ \Pi_i f_i = 1\ \mathsf{then}\ r\ \mathsf{else}\ \mathsf{id}\end{array}$$

A *consolidation stage* is the sum of zero or more predicated *multicast sequences*, each of which assigns to a set of fields (used for tagging each packet copy for later processing). We write $\Pi_i f_i \leftarrow v_i$ to stand for a sequence of field modifications $f_1 \leftarrow v_1; \ldots; f_n \leftarrow v_n$. Sequences may optionally begin with tables, which allows for multicast to be increased or decreased at run time. A consolidated mulicast stage ms may be sequenced with an *egress stages* n: some multicast-free policy fragments (r) made up of positive if statements predicated on multicast tags, and tables.

Definition 3 (Multicast consolidation)**.**

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

Given an arbitrary policy p, the function pipeline $s\ p$ factors the policy into a consolidation stage m followed by an egress stage n. The argument s provides the number of occurrences of multicast each table supports. The pipeline function is syntax-directed and presented in its entirety in the technical appendix; we highlight two

interesting cases here.

$$\begin{array}{l}\mathsf{pipeline}\ s\ (p+q) = \\ \quad \mathsf{let}\ f = \text{a fresh metadata field}\ \mathsf{in} \\ \quad \mathsf{let}\ (\sum_i m_i), n_1, \theta_1 = \mathsf{pipeline}\ s\ p\ \mathsf{in} \\ \quad \mathsf{let}\ (\sum_j m_j), n_2, \theta_2 = \mathsf{pipeline}\ s\ q\ \mathsf{in} \\ \quad \mathsf{let}\ n_3, \theta_3 = \mathsf{qualify}(f = 0, n_1)\ \mathsf{in} \\ \quad \mathsf{let}\ n_4, \theta_4 = \mathsf{qualify}(f = 1, n_2)\ \mathsf{in} \\ \quad ((\sum_i m_i; f \leftarrow 0) + (\sum_j m_j; f \leftarrow 1), n_3; n_4, \\ \qquad \theta_1 \circ \theta_2 \circ \theta_3 \circ \theta_4)\end{array}$$

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. Then, pick a fresh field f that neither p nor q use. For each summand in the consolidation stage produced from p, set f to 0, and assign 1 to f in summands produced from q. Finally, predicate each egress pipeline from p with $f = 0$ and from q with $f = 1$—the quantify function transforms if a then n else id into an egress pipeline n' with the predicate a flattened into each subseqent if statement. Finally, note that 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.

$$\begin{array}{l}\mathsf{pipeline}\ s\ (x:\tau) = \\ \quad \mathsf{let}\ fs = s(x)\ \text{fresh metadata fields}\ \mathsf{in} \\ \quad \mathsf{let}\ t_m = y : (\{\}, fs)\ \mathsf{in} \\ \quad \mathsf{let}\ t_n = z : (\tau.1 \cup fs, \tau.2)\ \mathsf{in} \\ \quad \mathsf{let}\ \theta' = (\lambda B, w.\mathsf{let}\ m, n, \theta = \mathsf{pipeline}\ s\ B(x)\ \mathsf{in} \\ \qquad \mathsf{if}\ w = y\ \mathsf{then}\ m\ \mathsf{else}\ \mathsf{if}\ w = z\ \mathsf{then}\ n\ \mathsf{else}\ T_{\theta\ B}\ w)\ \mathsf{in} \\ \quad (t_m, t_n, \theta')\end{array}$$

Pipelining a table variable produces a θ function that, in turn, compiles all future table updates—using the s map to preallocate metadata fields for future updates. A key property of table updates is that they produce closed terms—hence, when we invoke pipeline inside θ on the updated table $B(x)$, we run no risk of divergence.

Lemma 6 (Multicast consolidation preserves semantics)**.** *Let fs be the metadata fields used to tag multicast packets, and let $z = \Pi_{f \in fs} f \leftarrow 0$. If $\vdash p : \tau$ and $m, n, \theta = \mathsf{pipeline}\ s\ p$, then $T_b\ (z; p; z) \equiv T_{\theta\ b}\ (z; m; n; z)$.*

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. The proof proceeds by induction on the structure of the policy p.

6.2 Field extraction

The RMT model 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 a given field modification may take place, and then rewrites the policy so that modifications to that field happen first. For example, suppose we wish to extract modifications to the field f from the following policy.

$$\mathsf{if}\ b\ \mathsf{then}\ f \leftarrow v_1; p\ \mathsf{else}\ f \leftarrow v_2; q$$

Either f is set to v_1 or v_2, and the predicate b determines which occurs. Using a fresh field f', we can rewrite this policy to:

$$(b; f \leftarrow v_1; f' \leftarrow 0 + \neg b; f \leftarrow v_2; f' \leftarrow 1); \mathsf{if}\ f' = 0\ \mathsf{then}\ p\ \mathsf{else}\ q.$$

The field f' is necessary because b may depend on the value of f.

We define a *modification stage* as a sum of all the conditions leading to a given field being modified coupled with the modification. The function $\mathsf{ext}_f\ p$ splits a policy p into a modification stage for the field f followed by the remainder of the policy.

Definition 4 (Modification stage)**.**

$$\begin{array}{llll} \textit{modification sequence} & s & ::= & \Pi_i f_i \leftarrow v_i \\ & & \mid & (x:\tau); \Pi_i f_i \leftarrow v_i \\ \textit{modification sum} & e & ::= & \sum_j a_j; s_j \end{array}$$

Definition 5 (Field extraction)**.**

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

The interesting case lies in extracting modification conditions from within an if statement.

$$\begin{array}{l}\mathsf{ext}_f\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q) = \\ \quad \mathsf{let}\ f' = \text{a fresh metadata field}\ \mathsf{in} \\ \quad \mathsf{let}\ (\sum_i a_{1i}; m_{1i}); p_1, \theta_1 = \mathsf{ext}_f\ p\ \mathsf{in} \\ \quad \mathsf{let}\ (\sum_j a_{2j}; m_{2j}); q_2, \theta_2 = \mathsf{ext}_f\ q\ \mathsf{in} \\ \quad \mathsf{let}\ e = \sum_i b; a_{1i}; m_{1i}; f' \leftarrow 0 + \\ \qquad \sum_j \neg b; a_{2j}; m_{2j}; f' \leftarrow 1\ \mathsf{in} \\ \quad (e; \mathsf{if}\ f' = 0\ \mathsf{then}\ p_1\ \mathsf{else}\ q_2, \theta_2 \circ \theta_1)\end{array}$$

In this case, we begin by recursively extracting any modifications from the branches of the if statement. We then sequence the predicate b with the conditions produced from the true branch and $\neg b$ with those from the false branch. However, modifications m_{1i} (from the recursive call $\mathsf{ext}_f\ p$) or m_{2j} (from the recursive call $\mathsf{ext}_f\ q$) might affect the predicate b. We therefore save b's pre-modification value in a fresh field f'. After we've run the modification sums from p and q, we produce a conditional that now tests f', which holds the original result of the predicate b.

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.

Lemma 7 (Field extraction preserves semantics)**.** *Let fs be the metadata fields used to tag field extraction, and let $z = \Pi_{f \in fs} f \leftarrow 0$. If $\vdash r : \tau$ and $e, r', \theta = \mathit{pipeline}\ s\ r$, then $T_b\ (z; r; z) \equiv T_{\theta\ b}\ (z; e; r'; z)$.*

Composing multicast consolidation with field extraction (on the out field) produces two large summations. The next step is to factor the summations and group summands by output port. It is unclear whether/how the RMT architecture supports emitting multiple copies of a packet out the same output port, and so we reject programs of that shape here. Valid policies will now consist of a single large summation of field tests followed by modifications, ending with the modification of the out field.

$$\sum_i \Pi_j f_{ij} = v_{ij}; \Pi_k f_{ik} \leftarrow v_{ik}; \mathsf{out} \leftarrow i$$

The following transformation splits this summation into three. However, as we would have rejected the policy were it to send multiple packets out a single port, the first and last summations do not, in fact, ever emit multiple packets.

$$\begin{array}{l}(\sum_i \Pi_j f_{ij} = v_{ij}; out_i \leftarrow 1); \\ (\sum_i out_i = 1; f_{\mathsf{tag}} \leftarrow i; \mathsf{out} \leftarrow i); \\ (\sum_i f_{\mathsf{tag}} = i; \Pi_k f_{ik} \leftarrow v_{ik})\end{array}$$

We have not yet proved that this transformation is semantics preserving, although we expect that doing so is straightforward. The next section presents techniques for compiling these, and other policies, to physical table format.

6.3 Table fitting

With a few small modifications, we can adapt the compilation algorithm described in [3] for compiling policies to a physical table format. We call this *single table* compilation. But this style compilation comes with a cost—the number of rules in the compiled

table grows exponentially with the number of parallel and sequential combinators in the original policy. However, thanks to the concurrency inherent within 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, $(x_1 : \tau); (x_2 : \tau)$. We could compile $p; q$ into one table and place it into x_1 and r into x_2, or place p in x_1 and $q \,||\, r$ combined into x_2. If p is small and q and r are large, then compiling $p; q$ may be more efficient, as $q \,||\, r$ might be too big to fit in x_2 while p leaves space in x_1. The RMT chip has a limited number of bits a table can match and the number of rules it can hold—each match stage 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 fit function that uses a dynamic programming algorithm to search for the best one. The fit function takes a set of table annotations (t)—indicating the maximum number of rules that can be expected to be installed on each table—and a policy p, and produces a new policy consisting of the smallest sequence of table variables the policy can fit into, along with a binding transformation θ that describes how to compile run-time updates into the table sequence. The algorithm is $\mathcal{O}\left(n^3\right)$, where n is the number of atomic policy fragments in p.

7. Related work

NetCore [7, 11, 12] is a simple compositional language for specifying static data plane forwarding policy. NetKAT [3] extended NetCore with Kleene star, and a sound and complete equational theory for reasoning about networks. Concurrent NetCore shares a common core with NetCore (and NetKAT), but adds table specifications, concurrency, and a type system. These additions necessitate 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.

Concurrent Kleene Algebra (CKA) [8] is a related calculus that latter offers four composition operators: sequential composition, alternation, disjoint parallel composition and fine-grained concurrent composition. One key difference between NetCore/KAT and CKA (as well as other interpretations of Kleene algebra we are aware of) is that NetCore interprets "alternation" (disjunction) in a non-standard way as "copying parallel" composition. This leads to new and interesting interactions with our concurrent composition, which is most similar to CKA's disjoint parallel composition. Concurrent NetCore also has a type system and interpretation specialized to network programming, while CKA is presented at an extremely high level of abstraction.

Bossart *et al.* [4] recently proposed an architecture for programming OpenFlow 2.0 switches, which we follow in this paper. 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 [2].

8. Conclusion

Concurrent NetCore offers at once (a) a language for specifying routing policies and (b) packet-processing pipelines. It's novel operational semantics and type system recover strong reasoning principles. As such, it is an excellent intermediate language for compiling routing policies—since CNC can express both high-level policies and low-level pipelines, a multipass compiler can use the same reasoning principles throughout.

Acknowledgments

This work stemmed from discussions with Nick Feamster, Muhammad Shahbaz, and Jennifer Rexford. We would also like to thank Pat Bossart, Dan Daly, Glen Gibb, Nick McKeown, Dan Talayco, Amin Vahdat, and George Varghese for their insights on this topic.

References

[1] Concurrent netcore: From policies to pipelines. See `http://tinyurl.com/k2z8lz5`.

[2] Openflow forwarding abstractions working group charter, April 2013. See `http://goo.gl/TtLtw0`.

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

[4] 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`, December 2013.

[5] 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.

[6] Broadcom BCM56846 StrataXGS 10/40 GbE switch. See `http://www.broadcom.com/products/features/BCM56846.php`, 2014.

[7] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, September 2011.

[8] C. A. R. Hoare, Bernhard M oller, Georg Struth, and Ian Wehrman. Concurrent kleene algebra. In *CONCUR*, pages 399–414, 2009.

[9] 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.

[10] 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.

[11] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, January 2012.

[12] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, April 2013.

[13] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[14] Recep Ozdag. Intel Ethernet Switch FM6000 Series - software defined networking. See `goo.gl/AnvOvX`, 2012.

[15] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *HotSDN*, pages 43–48, 2012.

[16] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.