NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson
Swarthmore College *

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts Amherst *

Jean-Baptiste Jeannin
Carnegie Mellon University *

Dexter Kozen
Cornell University

Cole Schlesinger
Princeton University

David Walker
Princeton University

Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation has left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKAT, including primitives for filtering, modifying, and transmitting packets; union and sequential composition operators; and a Kleene star operator that iterates programs. We show that NetKAT is an instance of a canonical and well-studied mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking reachability, proving non-interference properties that ensure isolation between programs, and establishing the correctness of compilation algorithms.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

Keywords Software-defined networking, Frenetic, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKAT.

1. Introduction

Traditional network devices have been called "the last bastion of mainframe computing" [9]. Unlike modern computers, which are implemented with commodity hardware and programmed using standard interfaces, networks have been built the same way since the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, a revolution has taken place with the recent 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 new connections from hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly. Because the controller has a global view of the network, it is easy to use SDN to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault-tolerance.

A major appeal of SDN is that it defines open standards that any vendor can implement. For example, the OpenFlow API [21] clearly specifies the capabilities and behavior of switch hardware and defines a low-level language for manipulating their configurations. However, programs written directly for SDN platforms such as OpenFlow are akin to assembly: easy for hardware to implement, but difficult for humans to write.

Network programming languages. In recent years, several different research groups have proposed domain-specific languages for SDN [5–7, 23–25, 31, 32]. The goal of these *network programming languages* is to raise the level of abstraction of network programs above hardware-oriented APIs such as OpenFlow, thereby making it easier to build sophisticated and reliable SDN applications. For example, the languages developed in the Frenetic project [30] support a two-phase programming model: (i) a general-purpose program responds to network events by generating a static forwarding policy; and (ii) the static policy is compiled and passed to a run-time system that configures the switches using OpenFlow messages. This model balances expressiveness—dynamic policies can be expressed by having the general-purpose program generate a sequence of static policies—and simplicity—forwarding policies are written in a simple domain-specific language with a clear semantics, so programs can be analyzed and even verified using automated tools [7, 26].

Still, it has never been clear what features a static policy language should support. The initial version of Frenetic [6] used simple lists of predicate-action rules as policies, where the actions in-

* This work performed at Cornell University.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2535838.2535862

cluded constructs for filtering, forwarding, duplicating, and modifying packets. Subsequent versions of the language added (and later removed) the ability to embed arbitrary packet-processing functions in policies [23], as well as constructs for composing policies in parallel and sequence [24]. As Frenetic evolved, its designers added, removed, and modified the meaning of primitives as dictated by the needs of applications. Without principles or metatheory to guide its development, its evolution has lacked clear direction and foresight. The ad hoc semantics has not made clear which primitives are essential and which ones can be derived, and when new constructs have been added to the language, it has not been clear how they should interact with existing constructs and what behavioral laws they should obey.

An even more pressing issue is that these static policy languages only specify the forwarding behavior of the switches in the network. However, when a network program is actually executed, end-to-end functionality is determined both by the behavior of the switches and by the structure of the network topology. To answer almost any interesting question about the network such as *"Can X connect to Y?"*, *"Is traffic from A to B routed through Z?"*, or *"Is there a loop involving S?"*, the programmer must step outside the confines of the linguistic model and the abstractions it provides.

To summarize, we believe that a foundational model for network programming languages is essential. Such a model should (i) identify the essential constructs for programming networks, (ii) provide guidelines for incorporating new features, and (iii) unify reasoning about switches, topology and end-to-end behavior.

Semantic foundations. This paper presents the first network programming language that meets these criteria. To begin, we focus on the global behavior of the network, unlike previous network programming languages, which have focused on the local behavior of individual switches. Abstractly, a network can be seen as an automaton that moves packets from node to node along the links in its topology. Hence, from a linguistic perspective, it is natural to use regular expressions, the language of finite automata. Regular expressions are a standard way to specify the packet-processing behavior of a network: a path is encoded as a concatenation of processing steps ($p \cdot q \cdot \cdots$), a set of paths is encoded as a union of paths ($p + q + \cdots$), and iterated processing is encoded using Kleene star. Moreover, by modeling the network 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) [14].

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 compiler, device driver, and communication protocol verification [3, 15, 16, 22]. Moreover, equivalence in KAT has a PSPACE decision procedure. This paper applies this theory to a new domain: networks.

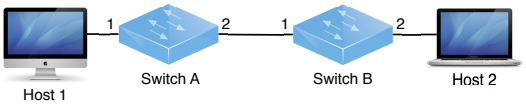

Figure 1. Example network.

NetKAT. NetKAT is a new framework for specifying, programming, and reasoning about networks based on Kleene algebra with tests. As a programming language, NetKAT has a simple denotational semantics inspired by NetCore [23], 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. Moreover, any future proposed primitive that violates a KAT axiom can be summarily rejected for breaking the equations that allow us to reason effectively about the network. NetKAT thus provides a foundational structure and consistent reasoning principles that other network programming languages lack.

For specification and reasoning, NetKAT also provides 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 transformations on packets. This set of axioms enables 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). We prove that the equational theory is sound and complete with respect to the denotational semantics. While the soundness proof is straightforward, our proof of completeness is novel: we construct an alternate language model for NetKAT and leverage the completeness of KA.

To evaluate the practical utility of our theory and the expressive power of NetKAT, we demonstrate how it can be used to reason about a diverse collection of applications. First, we show that NetKAT can answer a variety of interesting reachability queries useful to network operators. Next, we state and prove a non-interference property for networks that provides a strong form of isolation between NetKAT programs. Finally, we prove that NetKAT can be correctly compiled to a low-level form analogous to switch flow tables.

In summary, the contributions of this paper are as follows:

- We develop a new semantic foundation for network programming languages based on Kleene algebra with tests (KAT).
- We formalize the NetKAT language in terms of a denotational semantics and an axiomatic semantics based on KAT; we prove the equational axioms sound and complete with respect to the denotational semantics.
- We apply the equational theory in several diverse domains including reasoning about reachability, traffic isolation, and compiler correctness.

The next section presents a simple example to motivate NetKAT and introduces the key elements of its design. The subsequent sections define the language formally, develop its main theoretical properties, and present applications.

2. Overview

This section introduces the syntax and semantics of NetKAT using a simple example. Consider the network shown in Figure 1. It consists of switches A and B, each with 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 implement the following policies:

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

The forwarding component is straightforward—configure both switches to forward packets destined for host 1 out port 1, and likewise for host 2—but there are several ways to implement the access control component. We will develop two implementations and prove them equivalent using NetKAT's equational theory.

Forwarding. To warm up, let us define a simple NetKAT policy that implements the forwarding component. 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 a packet as a record 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 current location of the packet in the network.

Atomic NetKAT policies filter and modify packets. A filter $(f = n)$ takes any input packet pk and yields the singleton set $\{pk\}$ if field f of pk equals n, and $\{\}$ otherwise. A modification $(f \leftarrow n)$ takes any input packet pk and yields the singleton set $\{pk'\}$, where pk' is the packet obtained from pk by setting f to n.

To allow programmers to express more sophisticated policies, NetKAT also has policy combinators that build bigger policies out of smaller ones. The union combinator $(p + q)$ generates the union of the sets produced by applying each of p and q to the input packet, while the sequential composition combinator $(p \cdot q)$ first applies p to the input packet, then applies q to each packet in the resulting set, and finally takes the union of all of the resulting sets. With these operators, we can implement the forwarding policy as follows:

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

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 union of their behaviors—in other words, the policy forwards packets across switches A and B 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 in sequence:

$$p_{AC} \triangleq \neg(\mathsf{typ} = \text{SSH}) \cdot p$$

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 not necessary to test *all* packets at *all* locations in the network to block SSH traffic—packets traveling between host 1 and host 2 must traverse both switches, so it is sufficient to filter only at switch A,

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

or at switch B:

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

Both of these policies are more complicated than the original policy, but more efficient because they avoid having to store and enforce the access control policy at both switches. Naturally, we would prefer one of the optimized policies. In addition, we would like to be able to answer the following questions:

- *"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 write a network policy. However, note that we cannot answer them by inspecting the policies alone—the answers depend fundamentally on the network topology. We will see how to incorporate topology information into a NetKAT program next.

Topology. A network topology is a directed graph with hosts and switches as nodes and links as edges. We can model the topology as the union of smaller policies that encode the behavior of each link. To model an internal link, we use 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. To model a link at the perimeter of the network, we simply use a filter that retains packets located at the ingress port. We assume that links are uni-directional, and encode bi-directional links using pairs of uni-directional links. For example, the following policy models the internal links between switches A and B, and the links at the perimeter to hosts 1 and 2:

$$\begin{aligned} t = {} & (\mathsf{sw} = A \cdot \mathsf{pt} = 2 \cdot \mathsf{sw} \leftarrow B \cdot \mathsf{pt} \leftarrow 1) + \\ & (\mathsf{sw} = B \cdot \mathsf{pt} = 1 \cdot \mathsf{sw} \leftarrow A \cdot \mathsf{pt} \leftarrow 2) + \\ & (\mathsf{sw} = A \cdot \mathsf{pt} = 1) + \\ & (\mathsf{sw} = B \cdot \mathsf{pt} = 2) \end{aligned}$$

Note that although we represent the links as policies, unlike switch policies, these link policies cannot actually be controlled programmatically—they must be consistent with the structure of the underlying physical topology.

Switches meet topology. A packet traverses the network in interleaved steps of processing by the switches and topology. 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 $p_{AC} \cdot t \cdot p_{AC}$. More generally, a packet may require an arbitrary number of steps—in particular, if the topology has a cycle. Using the Kleene star operator, which iterates a policy zero or more times, we can encode the overall behavior of the network:

$$(p_{AC} \cdot t)^*$$

Note however that this policy processes packets that enter and exit the network at arbitrary locations, including at internal locations such as on the link between switches A and B. It is often useful to restrict attention to packets that enter and exit the network at specified external locations e:

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

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

$$p_{net} \triangleq e \cdot (p_{AC} \cdot t)^* \cdot e$$

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

$$in \cdot (p \cdot t)^* \cdot out$$

This encoding is inspired by the model used in Header Space Analysis [10]. We call a network modeled in this way a *logical crossbar* [20], since it encodes end-to-end processing behavior (and elides internal processing steps). Section 3 discusses a more refined model that encodes hop-by-hop processing.

Formal reasoning. We now turn to formal reasoning and investigate whether the logical crossbar correctly implements the specified forwarding and access control policies. It turns out that these

Syntax

Fields	f	$::=$	$f_1 \mid \cdots \mid f_k$
Packets	pk	$::=$	$\{f_1 = v_1, \cdots, f_k = v_k\}$
Histories	h	$::=$	$pk{::}\langle\rangle \mid pk{::}h$
Predicates	a, b	$::=$	$\mathsf{1}$ *Identity*
		$\mid$	$\mathsf{0}$ *Drop*
		$\mid$	$f = n$ *Test*
		$\mid$	$a + b$ *Disjunction*
		$\mid$	$a \cdot b$ *Conjunction*
		$\mid$	$\neg a$ *Negation*
Policies	p, q	$::=$	a *Filter*
		$\mid$	$f \leftarrow n$ *Modification*
		$\mid$	$p + q$ *Union*
		$\mid$	$p \cdot q$ *Sequential composition*
		$\mid$	p^* *Kleene star*
		$\mid$	dup *Duplication*

Semantics

$$[\![p]\!] \in \mathsf{H} \to \mathcal{P}(\mathsf{H})$$

$$[\![\mathsf{1}]\!]\ h \triangleq \{h\}$$

$$[\![\mathsf{0}]\!]\ h \triangleq \{\}$$

$$[\![f = n]\!]\ (pk{::}h) \triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$$[\![\neg a]\!]\ h \triangleq \{h\} \setminus ([\![a]\!]\ h)$$

$$[\![f \leftarrow n]\!]\ (pk{::}h) \triangleq \{pk[f := n]{::}h\}$$

$$[\![p + q]\!]\ h \triangleq [\![p]\!]\ h \cup [\![q]\!]\ h$$

$$[\![p \cdot q]\!]\ h \triangleq ([\![p]\!] \bullet [\![q]\!])\ h$$

$$[\![p^*]\!]\ h \triangleq \bigcup_{i \in \mathbb{N}} F^i\ h$$

where $F^0\ h \triangleq \{h\}$ and $F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h$

$$[\![\mathsf{dup}]\!]\ (pk{::}h) \triangleq \{pk{::}(pk{::}h)\}$$

Kleene Algebra Axioms

$p + (q + r) \equiv (p + q) + r$	KA-PLUS-ASSOC
$p + q \equiv q + p$	KA-PLUS-COMM
$p + \mathsf{0} \equiv p$	KA-PLUS-ZERO
$p + p \equiv p$	KA-PLUS-IDEM
$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$	KA-SEQ-ASSOC
$\mathsf{1} \cdot p \equiv p$	KA-ONE-SEQ
$p \cdot \mathsf{1} \equiv p$	KA-SEQ-ONE
$p \cdot (q + r) \equiv p \cdot q + p \cdot r$	KA-SEQ-DIST-L
$(p + q) \cdot r \equiv p \cdot r + q \cdot r$	KA-SEQ-DIST-R
$\mathsf{0} \cdot p \equiv \mathsf{0}$	KA-ZERO-SEQ
$p \cdot \mathsf{0} \equiv \mathsf{0}$	KA-SEQ-ZERO
$\mathsf{1} + p \cdot p^* \equiv p^*$	KA-UNROLL-L
$q + p \cdot r \leq r \Rightarrow p^* \cdot q \leq r$	KA-LFP-L
$\mathsf{1} + p^* \cdot p \equiv p^*$	KA-UNROLL-R
$p + q \cdot r \leq q \Rightarrow p \cdot r^* \leq q$	KA-LFP-R

Additional Boolean Algebra Axioms

$a + (b \cdot c) \equiv (a + b) \cdot (a + c)$	BA-PLUS-DIST
$a + \mathsf{1} \equiv \mathsf{1}$	BA-PLUS-ONE
$a + \neg a \equiv \mathsf{1}$	BA-EXCL-MID
$a \cdot b \equiv b \cdot a$	BA-SEQ-COMM
$a \cdot \neg a \equiv \mathsf{0}$	BA-CONTRA
$a \cdot a \equiv a$	BA-SEQ-IDEM

Packet Algebra Axioms

$f \leftarrow n \cdot f' \leftarrow n' \equiv f' \leftarrow n' \cdot f \leftarrow n$, if $f \neq f'$	PA-MOD-MOD-COMM
$f \leftarrow n \cdot f' = n' \equiv f' = n' \cdot f \leftarrow n$, if $f \neq f'$	PA-MOD-FILTER-COMM
$\mathsf{dup} \cdot f = n \equiv f = n \cdot \mathsf{dup}$	PA-DUP-FILTER-COMM
$f \leftarrow n \cdot f = n \equiv f \leftarrow n$	PA-MOD-FILTER
$f = n \cdot f \leftarrow n \equiv f = n$	PA-FILTER-MOD
$f \leftarrow n \cdot f \leftarrow n' \equiv f \leftarrow n'$	PA-MOD-MOD
$f = n \cdot f = n' \equiv \mathsf{0}$, if $n \neq n'$	PA-CONTRA
$\sum_i f = i \equiv \mathsf{1}$	PA-MATCH-ALL

Figure 2. NetKAT: syntax, semantics, and equational axioms.

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 going from port 1 on switch A to port 2 on switch B, we check the following equivalence, where $\mathsf{0}$ is the filtering policy that drops all packets:

$$\left(\begin{array}{l} \mathsf{typ} = \text{SSH} \cdot \mathsf{sw} = A \cdot \mathsf{pt} = 1 \cdot \\ (p_{AC} \cdot t)^* \cdot \\ \mathsf{sw} = B \cdot \mathsf{pt} = 2 \end{array}\right) \equiv \mathsf{0}$$

To establish that the optimized policies p_A and p_B correctly filter SSH packets going from port 1 on switch A to port 2 on switch B, we check the following equivalences,

$$\left(\begin{array}{l} \mathsf{typ} = \text{SSH} \cdot \mathsf{sw} = A \cdot \mathsf{pt} = 1 \cdot \\ (p_A \cdot t)^* \cdot \\ \mathsf{sw} = B \cdot \mathsf{pt} = 2 \end{array}\right) \equiv \mathsf{0}$$

and:

$$\left(\begin{array}{l} \mathsf{typ} = \text{SSH} \cdot \mathsf{sw} = A \cdot \mathsf{pt} = 1 \cdot \\ (p_B \cdot t)^* \cdot \\ \mathsf{sw} = B \cdot \mathsf{pt} = 2 \end{array}\right) \equiv \mathsf{0}$$

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

$$\begin{array}{l} (\neg(\mathsf{typ} = \text{SSH}) \cdot \mathsf{sw} = A \cdot \mathsf{pt} = 1 \cdot \mathsf{sw} \leftarrow B \cdot \mathsf{pt} \leftarrow 2) \\ \leq (p_{AC} \cdot t)^* \end{array}$$

and similarly for non-SSH packets H_2 to H_1.

Of course, to actually check these equivalences formally, 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 (Section 3); prove that the equational theory is sound and complete with respect to the semantics (Section 4); and illustrate its effectiveness on a broad range of questions including additional reachability properties (Section 5), program isolation (Section 6) and compiler correctness (Section 7).

3. NetKAT

This section defines the syntax and semantics of NetKAT 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 source and destination addresses, VLAN tag, IP source and destination addresses, TCP and UDP source and destination 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 from pk by updating field f to 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 previously-processed 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) and policies (p, q). Predicates include constants true (1) and false (0), tests $(f = n)$, and negation $(\neg a)$, disjunction $(a{+}b)$, and conjunction $(a{\cdot}b)$ operators. Policies include predicates, modifications $(f \leftarrow n)$, union $(p + q)$ and sequential composition $(p \cdot q)$, 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. By convention, $(^*)$ binds tighter than $(\cdot)$, which binds tighter than $(+)$. Hence, $a \cdot b + c \cdot d^*$ is the same as $(a \cdot b) + (c \cdot (d^*))$.

Semantics. Semantically, every NetKAT predicate and policy denotes a function that takes a history h and produces a (possibly empty) set of histories $\{h_1, \ldots, h_n\}$. 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 that an implementation need not actually record histories—they are only needed for reasoning.

Figure 2 defines the denotational semantics of NetKAT. Note that there is no separate definition for predicates—every predicate is a policy, and the semantics of $(\cdot)$ 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 $\{\}$ 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. The union operator $(p + q)$ denotes a function that produces the union of the sets generated by p and q, and sequential composition $(p{\cdot}q)$ 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.

Readers familiar with Frenetic and NetCore may notice some differences between the syntax used in previous network programming languages. This paper focuses on theoretical foundations, so we use the traditional "algebraic" syntax for KAT, which conflates

$a, b ::=$	`a,b ::=`
1	`true`
0	`false`
$f = n$	`f = n`
$a + b$	`a or b`
$a \cdot b$	`a and b`
$\neg a$	`not a`
$p, q ::=$	`p,q ::=`
a	`filter a`
$f \leftarrow n$	`f := n`
$p + q$	`p | q`
$p \cdot q$	`p ; q`
p^*	`p*`
dup	`dup`

Figure 3. NetKAT algebraic and surface syntax.

the $(\cdot)$ and $(+)$ operators for predicates and programs and has constants 0 and 1. Figure 3 shows the relationship between this algebraic syntax and the surface syntax we use in programs.

Equational theory. As its name suggests, NetKAT is a 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. A *Kleene algebra with tests* (KAT) is a two-sorted algebraic structure,

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

where $\neg$ is 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 $(K,\ +,\ \cdot,\ 0,\ 1)$.

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. In previous work on KAT, the elements of B and K have usually been called *tests* and *actions* respectively; in this paper we will refer to them as predicates and policies.

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. The first three axioms specify commutativity conditions. For example, the axiom 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:

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

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 final two axioms characterize filters. The axiom PA-CONTRA states that a field cannot be equal to two different values at the same time, while the axiom

KAT-INVARIANT	If $a \cdot p \equiv p \cdot a$ then $a \cdot p^* \equiv a \cdot (p \cdot a)^*$	Lemma 2.3.2 in [14]
KAT-SLIDING	$p \cdot (q \cdot p)^* \equiv (p \cdot q)^* \cdot p$	Identity 19 in [14]
KAT-DENESTING	$p^* \cdot (q \cdot p^*)^* \equiv (p + q)^*$	Identity 20 in [14]
KAT-COMMUTE	If for all atomic x in q, $x \cdot p \equiv p \cdot x$ then $q \cdot p \equiv p \cdot q$	Corollary of Lemma 4.4 in [2]

Figure 4. KAT theorems.

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.

Example: access control. To illustrate the NetKAT equational theory, we prove a simple equivalence in Figure 5 using the policies from Section 2. Recall that the policy p_A filters SSH packets on switch A while p_B filters SSH packets on switch B. We prove that these programs are equivalent on SSH traffic going from left to right across the network topology shown in Figure 1. This can be seen as a simple form of code motion—relocating the filter from switch A to switch B. We use the logical crossbar encoding with the following input and output predicates:

$$in \triangleq (\mathsf{sw} = A \cdot \mathsf{pt} = 1)$$
$$out \triangleq (\mathsf{sw} = B \cdot \mathsf{pt} = 2)$$

As a warm up, we first prove two simpler lemmas that will be useful for the main code motion proof. The proofs of these lemmas are straightforward calculations using the NetKAT axioms and some standard KAT theorems (given in Figure 4). We shade the term(s) on each line that will be changed on the next step of the proof. To lighten the notation, we elide uses of axioms related to associativity, and use the following abbreviations:

$$a_A \triangleq (\mathsf{sw} = A) \qquad a_1 \triangleq (\mathsf{pt} = 1)$$
$$a_B \triangleq (\mathsf{sw} = B) \qquad a_2 \triangleq (\mathsf{pt} = 2)$$
$$m_A \triangleq (\mathsf{sw} \leftarrow A) \qquad m_1 \triangleq (\mathsf{pt} \leftarrow 1)$$
$$m_B \triangleq (\mathsf{sw} \leftarrow B) \qquad m_2 \triangleq (\mathsf{pt} \leftarrow 2)$$
$$\text{SSH} \triangleq (\mathsf{typ} = \text{SSH})$$

Lemma 1 states that the sequential composition of the input predicate, the predicate a_B, and an arbitrary policy q is equivalent to the policy that drops all packets. Intuitively this holds because the input predicate matches packets on switch A and a_B matches packets on switch B. Lemma 2 states that the sequential composition of an arbitrary policy q, the predicate a_A, the topology t from Section 2, and the output predicate is equivalent to the policy that drops all packets. Intuitively this holds because the topology does not forward packets located on any port of switch A to port 2 of switch B. The proof is essentially a case analysis on the links in the topology. Using these lemmas, we then prove Lemma 3, which states that p_A and p_B both drop SSH traffic going from host 1 to host 2. Formal statements of these lemmas and proofs using the NetKAT equational axioms can be found in Figure 5.

4. Soundness, Completeness, and Decidability

This section proves the soundness and completeness of the NetKAT axioms with respect to the denotational semantics defined in Section 3. More formally, these results state that every equivalence provable using the NetKAT axioms also holds in the denotational model (Theorem 1), and that every equivalence which holds in the denotational model is provable using the axioms (Theorem 2). We also prove the decidability of NetKAT equivalence, and show that the problem is PSPACE-complete.

To obtain these results, we prove theorems that are stronger and more enlightening from a theoretical point of view. For soundness, we prove that the packet-history model used in the denotational semantics is isomorphic to a model based on binary relations, and appeal to the soundness of KAT over binary relation models. For completeness, we develop a language model for NetKAT that plays the same role as regular sets of strings and guarded strings do for KA and KAT respectively. We then relate the packet-history and language models, which allows us to leverage the completeness of KA to prove the completeness of the NetKAT axioms.

4.1 Soundness

To prove soundness, we begin by reformulating the standard packet-history semantics introduced in Section 3 in terms of binary relations. In the standard semantics, policies and predicates are modeled as functions $[\![p]\!] \in \mathsf{H} \to \mathcal{P}(\mathsf{H})$. This semantics is isomorphic to a relational semantics $[\cdot]$ in which each policy and predicate is interpreted as a binary relation $[p] \subseteq \mathsf{H} \times \mathsf{H}$:

$$(h_1, h_2) \in [p] \;\Leftrightarrow\; h_2 \in [\![p]\!]\,(h_1).$$

Intuitively, $[p]$ is the set of input-output pairs of the policy p.

Formally, the maps $[\![p]\!] \in \mathsf{H} \to \mathcal{P}(\mathsf{H})$ are morphisms of type $\mathsf{H} \to \mathsf{H}$ in $\mathsf{Kl}\,\mathcal{P}$, the Kleisli category of the powerset monad. It is well known that the Kleisli category $\mathsf{Kl}\,\mathcal{P}$ is isomorphic to the category Rel of sets and binary relations, as witnessed by currying:

$$X \to \mathcal{P}(Y) \cong X \to Y \to \mathbf{2} \cong X \times Y \to \mathbf{2} \cong \mathcal{P}(X \times Y).$$

In the relational model $[\cdot]$, product is interpreted as ordinary relational composition, and the remaining KAT operations translate under the isomorphism to the usual KAT operations on binary relations. Since the relational model with these distinguished operations satisfies the axioms of KAT (see *e.g.* [14, 17]), so do NetKAT models with the packet-history semantics of Section 3.

Let $\vdash$ denote provability in NetKAT. The following Theorem states the soundness of the NetKAT axioms.

Theorem 1 (Soundness)**.** *The* KAT *axioms and packet algebra axioms listed in Figure 2 are sound with respect to the semantics of Section 3. That is, if* $\vdash p \equiv q$*, then* $[\![p]\!] = [\![q]\!]$.

Proof sketch. We have already argued that the packet-history semantics is isomorphic to a relational KAT, and therefore satisfies the KAT axioms listed in Figure 2. It remains to show that the additional packet algebra axioms on the right-hand side of Figure 2 are also satisfied. These can all be verified by elementary arguments in relational algebra (see *e.g.* [28]). Some are special cases of [2, Equations (6)–(11)], whose soundness is proved in [2, Theorem 4.3]. See the long version of this paper for the full proof. □

4.2 Completeness

The proof of completeness proceeds in four steps:

1. We first define *reduced* NetKAT, a subset of NetKAT where policies are regular expressions over *complete tests* (a normal form for sequences of tests), *complete assignments* (a normal form for sequences of modifications), and dup. We show that every NetKAT policy is provably equivalent to a reduced NetKAT policy and that reduced terms have a simplified set of axioms pertaining to assignments and tests.
2. Inspired by past proofs of completeness for KA and KAT, we then develop a *language model* for reduced NetKAT. This language model gives semantics to policies via sets of guarded

Lemma 1. $in \cdot a_B \cdot q \equiv 0$

Proof.

$$
\begin{aligned}
& in \cdot a_B \cdot q \\
\equiv\ & \{ \text{ definition } in \ \} \\
& a_A \cdot a_1 \cdot a_B \cdot q \\
\equiv\ & \{ \text{ KAT-Commute } \} \\
& a_A \cdot a_B \cdot a_1 \cdot q \\
\equiv\ & \{ \text{ PA-Contra } \} \\
& 0 \cdot a_1 \cdot q \\
\equiv\ & \{ \text{ KA-Zero-Seq } \} \\
& 0 \qquad \square
\end{aligned}
$$

Lemma 2. $q \cdot a_A \cdot t \cdot out \equiv 0$

Proof.

$$
\begin{aligned}
& q \cdot a_A \cdot t \cdot out \\
\equiv\ & \{ \text{ definition } t \ \} \\
& q \cdot a_A \cdot (a_A \cdot a_2 \cdot m_B \cdot m_1 + \\
& \qquad a_B \cdot a_1 \cdot m_A \cdot m_2 + \\
& \qquad a_A \cdot a_1 + \\
& \qquad a_B \cdot a_2) \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Dist-L, KA-Seq-Dist-R } \} \\
& q \cdot a_A \cdot a_A \cdot a_2 \cdot m_B \cdot m_1 \cdot out + \\
& q \cdot a_A \cdot a_B \cdot a_1 \cdot m_A \cdot m_2 \cdot out + \\
& q \cdot a_A \cdot a_A \cdot a_1 \cdot out + \\
& q \cdot a_A \cdot a_B \cdot a_2 \cdot out \\
\equiv\ & \{ \text{ definition } out \ \} \\
& q \cdot a_A \cdot a_A \cdot a_2 \cdot m_B \cdot m_1 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_1 \cdot m_A \cdot m_2 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_A \cdot a_1 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_2 \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ PA-Mod-Filter } \} \\
& q \cdot a_A \cdot a_A \cdot a_2 \cdot m_B \cdot m_1 \cdot a_1 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_1 \cdot m_A \cdot a_A \cdot m_2 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_A \cdot a_1 \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_2 \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ KAT-Commute } \} \\
& q \cdot a_A \cdot a_A \cdot a_2 \cdot m_B \cdot m_1 \cdot a_B \cdot a_1 \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_1 \cdot m_A \cdot m_2 \cdot a_A \cdot a_B \cdot a_2 + \\
& q \cdot a_A \cdot a_A \cdot a_B \cdot a_1 \cdot a_2 + \\
& q \cdot a_A \cdot a_B \cdot a_2 \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ PA-Contra } \} \\
& q \cdot a_A \cdot a_A \cdot a_2 \cdot m_B \cdot m_1 \cdot a_B \cdot 0 + \\
& q \cdot a_A \cdot a_B \cdot a_1 \cdot m_A \cdot m_2 \cdot 0 \cdot a_2 + \\
& q \cdot a_A \cdot a_A \cdot a_B \cdot 0 + \\
& q \cdot 0 \cdot a_2 \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ KA-Seq-Zero, KA-Zero-Seq } \} \\
& 0 + 0 + 0 + 0 \\
\equiv\ & \{ \text{ KA-Plus-Idem } \} \\
& 0 \qquad \square
\end{aligned}
$$

Lemma 3. $in \cdot \text{SSH} \cdot (p_A \cdot t)^* \cdot out \equiv in \cdot \text{SSH} \cdot (p_B \cdot t)^* \cdot out$

Proof.

$$
\begin{aligned}
& in \cdot \text{SSH} \cdot (p_A \cdot t)^* \cdot out \\
\equiv\ & \{ \text{ KAT-Invariant, definition } p_A \ \} \\
& in \cdot \text{SSH} \cdot ((a_A \cdot \neg\text{SSH} \cdot p + a_B \cdot p) \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Dist-R } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot \neg\text{SSH} \cdot p \cdot t \cdot \text{SSH} + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KAT-Commute } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot \neg\text{SSH} \cdot \text{SSH} \cdot p \cdot t + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ BA-Contra } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot 0 \cdot p \cdot t + a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Zero/Zero-Seq, KA-Plus-Comm, KA-Plus-Zero } \} \\
& in \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KA-Unroll-L } \} \\
& in \cdot \text{SSH} \cdot (1 + (a_B \cdot p \cdot t \cdot \text{SSH}) \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^*) \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Dist-L, KA-Seq-Dist-R, definition } out \ \} \\
& in \cdot \text{SSH} \cdot a_B \cdot a_2 + \\
& in \cdot \text{SSH} \cdot a_B \cdot p \cdot t \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ KAT-Commute } \} \\
& in \cdot a_B \cdot \text{SSH} \cdot a_2 + \\
& in \cdot a_B \cdot \text{SSH} \cdot p \cdot t \cdot \text{SSH} \cdot (a_B \cdot p \cdot t \cdot \text{SSH})^* \cdot a_B \cdot a_2 \\
\equiv\ & \{ \text{ Lemma 1 } \} \\
& 0 + 0 \\
\equiv\ & \{ \text{ KA-Plus-Idem } \} \\
& 0 \\
\equiv\ & \{ \text{ KA-Plus-Idem } \} \\
& 0 + 0 \\
\equiv\ & \{ \text{ Lemma 1, Lemma 2 } \} \\
& in \cdot a_B \cdot \text{SSH} \cdot a_2 + \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot p \cdot \text{SSH} \cdot a_A \cdot t \cdot out \\
\equiv\ & \{ \text{ KAT-Commute, definition } out \ \} \\
& in \cdot \text{SSH} \cdot out + \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot a_A \cdot p \cdot t \cdot \text{SSH} \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Dist-L, KA-Seq-Dist-R } \} \\
& in \cdot \text{SSH} \cdot (1 + (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot (a_A \cdot p \cdot t \cdot \text{SSH})) \cdot out \\
\equiv\ & \{ \text{ KA-Unroll-R } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Zero/Zero-Seq, KA-Plus-Zero } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot 0 \cdot p \cdot t)^* \cdot out \\
\equiv\ & \{ \text{ BA-Contra } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot \neg\text{SSH} \cdot \text{SSH} \cdot p \cdot t)^* \cdot out \\
\equiv\ & \{ \text{ KAT-Commute } \} \\
& in \cdot \text{SSH} \cdot (a_A \cdot p \cdot t \cdot \text{SSH} + a_B \cdot \neg\text{SSH} \cdot p \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KA-Seq-Dist-R } \} \\
& in \cdot \text{SSH} \cdot ((a_A \cdot p + a_B \cdot \neg\text{SSH} \cdot p) \cdot t \cdot \text{SSH})^* \cdot out \\
\equiv\ & \{ \text{ KAT-Invariant, definition } p_B \ \} \\
& in \cdot \text{SSH} \cdot (p_B \cdot t)^* \cdot out \qquad \square
\end{aligned}
$$

Figure 5. Code motion proofs.

strings. We prove the language model and the standard model of NetKAT given by the denotational semantics are isomorphic.

3. We then define a normal form for NetKAT policies, and show that every policy is provably equivalent to its normal form.
4. Finally, we relate NetKAT normal forms to regular sets of guarded strings, and obtain the completeness of NetKAT from the completeness of KA.

The rest of this section outlines the key steps of this proof. The long version of this paper gives further details.

Step 1: Reduced NetKAT. Let $f_1, \ldots, f_k$ be a list of all fields of a packet in some (fixed) order. For each tuple $\bar{n} = n_1, \ldots, n_k$ of values, let $\bar{f} = \bar{n}$ and $\bar{f} \leftarrow \bar{n}$ denote the expressions

$$f_1 = n_1 \cdots f_k = n_k \qquad f_1 \leftarrow n_1 \cdots f_k \leftarrow n_k,$$

Reduced NetKAT syntax

$$
\begin{array}{rrll}
\text{Complete assignments} & \pi & \triangleq & f_1 \leftarrow n_1 \cdots f_k \leftarrow n_k \\
\text{Complete tests} & \alpha, \beta & \triangleq & f_1 = n_1 \cdots f_k = n_k \\
\text{Reduced terms} & p, q & ::= & \alpha \quad \textit{Complete test} \\
 & & | & \pi \quad \textit{Complete assignment} \\
 & & | & p + q \quad \textit{Union} \\
 & & | & p \cdot q \quad \textit{Sequence} \\
 & & | & p^* \quad \textit{Kleene star} \\
 & & | & \mathsf{dup} \quad \textit{Duplication}
\end{array}
$$

Simplified axioms for A and P

$$\pi \equiv \pi \cdot \alpha_\pi \qquad \alpha \cdot \mathsf{dup} \equiv \mathsf{dup} \cdot \alpha \qquad \sum_\alpha \alpha \equiv 1,$$

$$\alpha \equiv \alpha \cdot \pi_\alpha \qquad \pi \cdot \pi' \equiv \pi' \qquad \alpha \cdot \beta \equiv 0,\ \alpha \neq \beta$$

Regular interpretation: $R(p) \subseteq (\Pi + A + \mathsf{dup})^*$

$$
\begin{aligned}
R(\pi) &= \{\pi\} \\
R(p+q) &= R(p) \cup R(q) \\
R(\alpha) &= \{\alpha\} \\
R(p \cdot q) &= \{xy \mid x \in R(p), y \in R(q)\} \\
R(\mathsf{dup}) &= \{\mathsf{dup}\} \\
R(p^*) &= \bigcup_{n \geq 0} R(p^n)
\end{aligned}
$$

Figure 6. Reduced NetKAT.

Language model: $G(p) \subseteq I = A \cdot (\Pi \cdot \mathsf{dup})^* \cdot \Pi$

$$
\begin{aligned}
G(\pi) &= \{\alpha \cdot \pi \mid \alpha \in A\} \\
G(p+q) &= G(p) \cup G(q) \\
G(\alpha) &= \{\alpha \cdot \pi_\alpha\} \\
G(p \cdot q) &= G(p) \diamond G(q) \\
G(\mathsf{dup}) &= \{\alpha \cdot \pi_\alpha \cdot \mathsf{dup} \cdot \pi_\alpha \mid \alpha \in A\} \\
G(p^*) &= \bigcup_{n \geq 0} G(p^n)
\end{aligned}
$$

Guarded concatenation

$$
\alpha \cdot p \cdot \pi \diamond \beta \cdot q \cdot \pi' = \begin{cases} \alpha \cdot p \cdot q \cdot \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases}
$$

$$A \diamond B = \{p \diamond q \mid p \in A,\ q \in B\} \subseteq I$$

Figure 7. NetKAT language model.

respectively. We call these expressions *complete tests* and *complete assignments* respectively. We often call complete tests *atoms* because they are atoms (minimal nonzero elements) of the Boolean algebra generated by the tests. Note that complete tests and complete assignments are in one-to-one correspondence according to the values $\bar{n}$. Hence, if α is an atom, we denote the corresponding complete assignment by π_α, and if π is a complete assignment, we denote the corresponding atom by α_π. We let A denote the set of atoms and Π the set of complete assignments.

Now that we have defined atoms and complete assignments, we investigate their properties. Figure 6 gives a collection of simple axioms for reduced policies that are easily provable using the full NetKAT axioms. One useful consequence of these axioms is

$$\textstyle\sum_{\alpha \in A} \alpha \cdot \pi_\alpha \equiv 1.$$

Any policy is provably equivalent to a policy in which all atomic assignments $f \leftarrow n$ appear in the context of a complete assignment. The proof of this fact is straightforward.

$$
\begin{aligned}
f \leftarrow n &\equiv 1 \cdot f \leftarrow n \\
&\equiv (\textstyle\sum_{\alpha \in A} \alpha \cdot \pi_\alpha) \cdot (f \leftarrow n) \\
&\equiv \textstyle\sum_{\alpha \in A} \alpha \cdot \pi'_\alpha
\end{aligned}
$$

where π'_α is π_α with the assignment to f replaced by $f \leftarrow n$. Similarly, every test is equivalent to a sum of complete tests:

$$b \equiv \textstyle\sum_{\alpha \leq b} \alpha$$

Since all modifications can be replaced by complete assignments and all tests by atoms, any NetKAT policy p can be viewed as a regular expression over the alphabet $\Pi \cup A \cup \{\mathsf{dup}\}$. The bottom of Figure 6 shows this by defining a mapping R from reduced NetKAT to regular sets over this alphabet. We assume for the remainder of this section that all NetKAT policies are in reduced form.

Step 2: Language model. Both KA and KAT have language models in which expressions are interpreted as regular sets of minimal nonzero terms (often called join-irreducible terms). For KA, the language model is the regular sets of strings, and for KAT, it is the regular sets of guarded strings [17]. NetKAT also has a language model. It consists of regular subsets of a restricted class of guarded strings $I = A \cdot (\Pi \cdot \mathsf{dup})^* \cdot \Pi$. Each string in this set has the form

$$\alpha \cdot \pi_0 \cdot \mathsf{dup} \cdot \pi_1 \cdot \mathsf{dup} \cdots \mathsf{dup} \cdot \pi_n$$

for some $n \geq 0$. These strings represent the minimal nonzero elements of the standard model of NetKAT.

Figure 7 defines the language model as a mapping G from reduced NetKAT expressions to regular subsets of I. The case for sequential composition makes use of the concatenation operator ($\diamond$) over strings from I, which we lift to concatenation of sets of guarded strings from I. Both definitions appear at the bottom of Figure 7. Note that $\diamond$ is a partial function on strings but a total function on sets of strings. Using the simplified axioms of Figure 6, it is easy to show that $\diamond$ is associative on strings and sets, distributes over union, and has two-sided identity $\{\alpha \cdot \pi_\alpha \mid \alpha \in A\}$. Also note that if

$$\alpha \cdot p \cdot \pi \diamond \beta \cdot q \cdot \pi'$$

exists, then

$$\vdash \alpha \cdot p \cdot \pi \cdot \beta \cdot q \cdot \pi' \equiv \alpha \cdot p \cdot \pi \diamond \beta \cdot q \cdot \pi' \in I$$

and otherwise:

$$\vdash \alpha \cdot p \cdot \pi \cdot \beta \cdot q \cdot \pi' \equiv 0$$

Having defined the language model, we now show that it is isomorphic to the standard packet model presented in Section 3. We first show that the standard semantics of every NetKAT expression is equal to the union of its minimal nonzero terms.[1] The proof is straightforward by induction on p.

Lemma 4. *For all policies p, we have $[\![p]\!] = \bigcup_{x \in G(p)} [\![x]\!]$.*

Next we prove that every x in I is completely determined by $[\![x]\!]$.

Lemma 5. *If $x, y \in I$, then $[\![x]\!] = [\![y]\!]$ if and only if $x = y$.*

Finally, using Lemmas 4 and 5, we conclude that the language model is isomorphic to the denotational model presented earlier.

Lemma 6. *For all policies p and q, we have $[\![p]\!] = [\![q]\!]$ if and only if $G(p) = G(q)$.*

[1] We abuse notation slightly here by applying the union operator $\bigcup$ to functions $\mathsf{H} \to \mathcal{P}(\mathsf{H})$. This is interpreted pointwise: $\bigcup [\![p]\!] = \lambda s . \bigcup [\![p]\!]\,(s)$.

Step 3: Normal forms. Next we define a normal form for NetKAT policies and prove that every policy is provably equivalent to one in normal form.

Definition 1. *A Net*KAT *policy p is in* normal form *if $R(p) \subseteq I$. A policy is* normalizable *if it is provably equivalent to a policy in normal form.*

Lemma 7. *Every policy p is normalizable.*

Proof. The inductive proof requires a slightly strengthened inductive hypothesis. Let us say that a policy is in *strong normal form* if it is in normal form and is a sum of zero or more guarded policies, where a policy is *guarded* if it is of the form either $\alpha \cdot \pi \cdot x \cdot \pi'$ or $\alpha \cdot \pi$. We show by induction on p that every policy is equivalent to a policy in strong normal form.

The cases for atomic policies are straightforward:

$$\begin{array}{lcl} h \leftarrow n & \equiv & \sum_{\alpha \in A} \alpha \cdot \pi'_\alpha \\ \mathsf{dup} & \equiv & \sum_{\alpha \in A} \alpha \cdot \pi_\alpha \cdot \mathsf{dup} \cdot \pi_\alpha \\ b & \equiv & \sum_{\alpha \leq b} \alpha \cdot \pi_\alpha \end{array}$$

The case for union is trivial, and the case for sequential composition follows by a simple argument:

$$\left(\sum_i s_i\right) \cdot \left(\sum_j t_j\right) \equiv \sum_i \sum_j s_i \cdot t_j \equiv \sum_i \sum_j s_i \diamond t_j.$$

The most interesting case is for Kleene star. Consider an expression p^*, where p is in strong normal form. We first prove the uniform case: when all guarded terms in p have the same initial atom α, that is, $p = \alpha \cdot t$ where t is a union of terms each with a leading and trailing π, and $R(t) \subseteq \Pi \cdot (\mathsf{dup} \cdot \Pi)^*$. Let u be t with all terms whose trailing π is not π_α deleted and with the trailing π_α deleted from all remaining terms. By the simplified axioms of Figure 6, we have $t \cdot \alpha \cdot t \equiv u \cdot t$, therefore $t \cdot \alpha \cdot t \cdot \alpha \equiv u \cdot t \cdot \alpha$. Using KAT-COMMUTE [2, Lemma 4.4],

$$\begin{array}{lcl} (t \cdot \alpha)^* \cdot t & \equiv & t + t \cdot \alpha \cdot (t \cdot \alpha)^* \cdot t \\ & \equiv & t + u^* \cdot t \cdot \alpha \cdot t \\ & \equiv & t + u^* \cdot u \cdot t \\ & \equiv & u^* \cdot t, \end{array}$$

and hence

$$\begin{array}{lcl} p^* & \equiv & 1 + p^* \cdot p \\ & \equiv & 1 + (\alpha \cdot t)^* \cdot \alpha \cdot t \\ & \equiv & 1 + \alpha \cdot (t \cdot \alpha)^* \cdot t \\ & \equiv & 1 + \alpha \cdot u^* \cdot t \\ & \equiv & 1 + \alpha \cdot t + \alpha \cdot u \cdot u^* \cdot t, \end{array}$$

which after normalizing the 1 is in strong normal form. For the case p^* where the initial tests in p are not uniform, the argument is by induction on the number of terms in the union. If $p = \alpha \cdot x + q$, then by the inductive hypothesis, q^* has an equivalent strong normal form $\hat{q}^*$. Using KAT-DENESTING (Figure 4), we obtain

$$p^* \equiv (\alpha \cdot x + q)^* \equiv q^* \cdot (\alpha \cdot x \cdot q^*)^* \equiv \hat{q}^* \cdot (\alpha \cdot x \cdot \hat{q}^*)^*,$$

then proceed as in the previous case. □

Step 4: Completeness. We need just one more lemma before delivering the completeness result, which says that the regular interpretation and language model coincide for NetKAT policies in normal form.

Lemma 8. *If $R(p) \subseteq I$, then $R(p) = G(p)$.*

Proof. Suppose $R(p) \subseteq I$. It is straightforward to show that $G(p)$ is equal to the union of the elements of $R(p)$, by induction on p:

$$G(p) = \bigcup_{x \in R(p)} G(x).$$

Then, since $G(x) = \{x\}$ for $x \in I$, we have

$$G(p) = \bigcup_{x \in R(p)} \{x\} = R(p). \qquad \square$$

The proof of completeness for NetKAT now follows from the completeness of KA [12].

Theorem 2 (Completeness)**.** *Every semantically equivalent pair of Net*KAT *expressions is provably equivalent using the Net*KAT *axioms. That is, if $[\![p]\!] = [\![q]\!]$, then $\vdash p \equiv q$.*

Proof. Let $\hat{p}$ and $\hat{q}$ be the normal forms of p and q. By Lemma 7, we can prove that each is equivalent to its normal form: $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$. By soundness we have $[\![p]\!] = [\![\hat{p}]\!]$ and $[\![q]\!] = [\![\hat{q}]\!]$, hence $[\![\hat{p}]\!] = [\![\hat{q}]\!]$. By Lemma 6, we have $G(\hat{p}) = G(\hat{q})$. Moreover, by Lemma 8, we have $G(\hat{p}) = R(\hat{p})$ and $G(\hat{q}) = R(\hat{q})$, thus $R(\hat{p}) = R(\hat{q})$. Since $R(\hat{p})$ and $R(\hat{q})$ are regular sets, we have $\vdash \hat{p} \equiv \hat{q}$ by the completeness of KA. Finally, as $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$ and $\vdash \hat{p} \equiv \hat{q}$, we conclude that $\vdash p \equiv q$. □

4.3 Decidability

The final theorem presented in this section shows that deciding the equational theory of NetKAT is no more nor less difficult than for KA or KAT.

Theorem 3. *The equational theory of Net*KAT *is* PSPACE*-complete.*

Proof sketch. To show PSPACE-hardness, reduce KA to NetKAT as follows. Let Σ be a finite alphabet. For a regular expression e over Σ, let $R(e)$ be the regular set of strings over Σ as defined in §4. Transform e to a NetKAT expression e' by replacing each occurrence in e of a symbol $p \in \Sigma$ with $(p \cdot \mathsf{dup})$ and prepending with an arbitrary but fixed atom α. It follows from Lemmas 6 and 8 that $R(e_1) = R(e_2)$ if and only if $R(e'_1) = R(e'_2)$ if and only if $G(e'_1) = G(e'_2)$ if and only if $[\![e'_1]\!] = [\![e'_2]\!]$.

To show that the problem is in PSPACE, given two NetKAT expressions e_1 and e_2, we know that $[\![e_1]\!] \neq [\![e_2]\!]$ if and only if there is a packet pk and packet history h such that $h \in [\![e_1]\!](pk) \setminus [\![e_2]\!](pk)$ or $h \in [\![e_2]\!](pk) \setminus [\![e_1]\!](pk)$; let us say the former without loss of generality. We guess pk nondeterministically and follow a nondeterministically-guessed trajectory through e_1 that produces some $h \in [\![e_1]\!](pk)$. At the same time, we trace all possible trajectories through e_2 that could generate a prefix of h, ensuring that none of these produce $h \in [\![e_2]\!]$. It takes only polynomial space to represent the current values of the fields of the head packet and the possible positions in e_2 for the current prefix of h. The algorithm is nondeterministic, but can be made deterministic using Savitch's theorem. □

5. Reachability Properties

Network administrators often ask questions such as, *"Can all hosts talk to each other?"*, or *"Are managed hosts kept separate from unmanaged hosts?"*, or *"Does all untrusted traffic traverse the intrusion detection system?"*, and so on. Automated tools for answering these and other questions about *reachability properties* have been the focus of several recent research projects [10, 11, 19]. Most existing tools work by encoding the topology and policy as a logical structure, and then translating the reachability property into a formula whose satisfiability can be checked using a SAT solver or other tool. This section presents a different approach: we show how to encode two important classes of reachability properties as NetKAT equations. We then prove the equations are sound and complete with respect to intuitive, semantic definitions of reachability using the language model developed in Section 4.

Reachability. The simplest reachability properties answer questions such as, *"Can host A send packets to host B?"* If A can send packets to B, then the denotational model of the network must include a packet history that starts from host A and ends at host B: $\langle pk_B, \cdots, pk_A \rangle$. More generally, we can ask if packets satisfying some predicate a can be transformed so that they satisfy some predicate b. These predicates may denote single hosts, groups of hosts, or even arbitrary classes of traffic, such as Web traffic.

To reason about reachability, we use a small generalization of the logical crossbar model introduced in Section 2. In the logical crossbar model, we model the end-to-end behavior of the network using the following NetKAT expression,

$$in \cdot (p \cdot t)^* \cdot out$$

where p and t define the behavior of switches and links as dup-free NetKAT policies. Because the policy does not contain dup, it does not record the individual "hops" that packets take as they go through the network. To do this, we must augment the logical crossbar to record the state of the packet at each intermediate hop:

$$in \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot out$$

Using this encoding, we can define reachability as follows:

Definition 2 (Reachability)**.** *We say b is reachable from a if and only if there exists a trace*

$$\langle pk_1, \cdots, pk_n \rangle \in \mathit{rng}([\![\mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^*]\!])$$

such that $[\![a]\!]\langle pk_n \rangle = \{\langle pk_n \rangle\}$ and $[\![b]\!]\langle pk_1 \rangle = \{\langle pk_1 \rangle\}$.

To decide whether b is reachable from a we check the following NetKAT equivalence:

$$a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b \not\equiv \mathsf{0}$$

Intuitively, the prefix predicate, a, filters the policy to only include histories that begin with packets satisfying a. Similarly, the postfix predicate, b, filters the policy to only include histories that end with packets satisfying b. We do need to prove that this equation holds if and only if b is reachable from a. The key to the proof is to translate both the denotational definition of reachability and the reachability equation to the language model, where they are easy to relate.

Theorem 4 (Reachability Correctness)**.** *For predicates a and b, policy p, and topology t, $a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b \not\equiv \mathsf{0}$, if and only if b is reachable from a.*

Proof. We translate the NetKAT equation into the language model:

$$\begin{array}{ll} & a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b \not\equiv \mathsf{0} \\ \Rightarrow & \exists \alpha, \pi_n, \cdots, \pi_1. \\ & \quad \alpha \cdot \pi_n \cdot \mathsf{dup} \cdots \mathsf{dup} \cdot \pi_1 \in G(a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b) \end{array}$$

We also translate each term in the semantic definition of reachability into the language model:

$$\begin{array}{ll} & \exists pk_1, \cdots, pk_n. \\ & \quad \langle pk_1, \cdots, pk_n \rangle \in \mathsf{rng}([\![\mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^*]\!]), \\ & \quad [\![a]\!]\langle pk_n \rangle = \{\langle pk_n \rangle\} \text{ and} \\ & \quad [\![b]\!]\langle pk_1 \rangle = \{\langle pk_1 \rangle\} \\ \Rightarrow & \exists \pi'_1, \cdots, \pi'_m. \\ & \quad \alpha_{\pi'_m} \cdot \pi'_m \cdot \mathsf{dup} \cdots \mathsf{dup} \cdot \pi'_1 \in G(\mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^*), \\ & \quad \alpha_{\pi'_m} \cdot \pi'_m \in G(a) \text{ and} \\ & \quad \alpha_{\pi'_1} \cdot \pi'_1 \in G(b) \end{array}$$

To prove soundness we let $\alpha = \alpha_{\pi_n}$ and $m = n$ to show that if

$$\alpha \cdot \pi_n \cdot \mathsf{dup} \cdots \mathsf{dup} \cdot \pi_1 \in G(a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b)$$

then,

$$\alpha_{\pi'_m} \cdot \pi'_m \cdot \mathsf{dup} \cdots \mathsf{dup} \cdot \pi'_1 \in G(\mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^*)$$

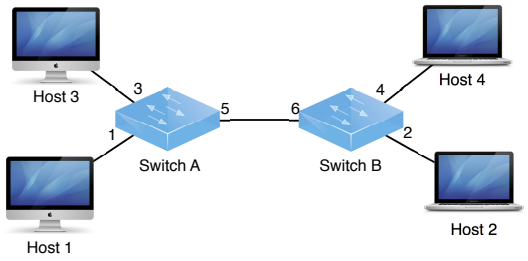

Figure 8. A simple network controlled by two parties.

which holds by definition of $\diamond$. The proof of completeness follows by a similar argument. □

Waypointing. A *waypoint*, w, from a to b is a location that all packets traverse en route from a to b. For example, a network operator might want to ensure that all traffic from untrusted hosts to trusted hosts traverses a firewall.

Definition 3 (Waypoint)**.** *We say w is a waypoint from a to b, if and only if, for all histories $\langle pk_1 \cdots pk_n \rangle \in \mathit{rng}([\![\mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^*]\!])$ where $[\![a]\!]\langle pk_n \rangle = \{\langle pk_n \rangle\}$ and $[\![b]\!]\langle pk_1 \rangle = \{\langle pk_1 \rangle\}$, there exists a $pk_x \in \langle pk_1 \cdots pk_n \rangle$ such that:*

- $[\![w]\!]\langle pk_x \rangle = \{\langle pk_x \rangle\}$,
- $[\![b]\!]\ pk_i = \{\}$ *for all* $1 < i < x$, *and*
- $[\![a]\!]\ pk_j = \{\}$ *for all* $x < j < n$.

To decide whether w is a waypoint from a to b, we check the following NetKAT inequality:

$$\begin{array}{l} a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b \\ \leq a \cdot \mathsf{dup} \cdot (\neg b \cdot p \cdot t \cdot \mathsf{dup})^* \cdot w \cdot (\neg a \cdot p \cdot t \cdot \mathsf{dup})^* \cdot b \end{array}$$

The left-hand side is exactly hop-by-hop reachability from a to b. The right-hand side is also a hop-by-hop expression, but it has a predicate to check that packets traverse w. Furthermore, it tests that packets do not prematurely visit b before w or return to a after reaching w.

Theorem 5 (Waypoint Correctness)**.** *For predicates a, b, and w,*

$$\begin{array}{l} a \cdot \mathsf{dup} \cdot (p \cdot t \cdot \mathsf{dup})^* \cdot b \\ \leq a \cdot \mathsf{dup} \cdot (\neg b \cdot p \cdot t \cdot \mathsf{dup})^* \cdot w \cdot (\neg a \cdot p \cdot t \cdot \mathsf{dup})^* \cdot b \end{array}$$

if and only if all packets from a to b are waypointed through w.

Proof. Similar to the proof of reachability correctness above. See the long version of this paper for the full proofs. □

Using these encodings and theorems as building blocks, we can develop techniques for checking other reachability properties as well. For example, we can check for self-loops, test whether a firewall policy is correctly implemented, and string together multiple waypoints into composite tests.

6. Traffic Isolation

NetKAT's policy combinators help programmers construct complex network policies out of simple parts. The most basic combinator is union, which combines two policies by taking the union of the results generated by the sub-policies. However, naive use of union can lead to undesirable results, because each sub-policy may receive and modify packets intended for the other sub-policy.

Example. Consider the network in Figure 8. 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,

$$p_{B1} \triangleq \mathsf{sw} = B \cdot (\mathsf{pt} = 6 \cdot \mathsf{pt} \leftarrow 2 + \mathsf{pt} = 2 \cdot \mathsf{pt} \leftarrow 6)$$

and the second programmer might produce a similar switch policy for B. This second policy differs from the first only by sending traffic from port 6 out port 4 rather than port 2:

$$p_{B2} \triangleq \mathsf{sw} = B \cdot (\mathsf{pt} = 6 \cdot \mathsf{pt} \leftarrow 4 + \mathsf{pt} = 4 \cdot \mathsf{pt} \leftarrow 6)$$

Similar policies p_{A1} and p_{A2} define the behavior at switch A. Now, if we assume t captures the topology of the network properly, then

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

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

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

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 unnecessary congestion. In the worst case, it may violate the security policy for host 1. Either alternative demonstrates the need for better ways of composing forwarding policies.

Slices. A *network slice* [8] is a lightweight abstraction that facilitates modular construction of policies. Intuitively, a slice defines a piece of the network that can be programmed independently of the rest of the network. The boundaries of a slice are defined by ingress (in) and egress (out) predicates, while the behavior in the slice is determined by the internal policy p. Each slice also has a unique identifier (w) to differentiate it from other slices.[2]

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.

To define slices in NetKAT, we begin by picking a header field, for example, tag, to record the packet's current slice.[3] 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:

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

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 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 policy of that one slice until it is ejected, even across multiple hops in the topology. The following theorem captures this idea more precisely.

Theorem 6 (Slice Composition)**.** *For all* tag*-free slice ingress and egress predicates in and out, identifiers w, policies $s, q,$* tag*-free policies p, and topologies t, if*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- $w \neq w_0$,
- $out \cdot t \cdot \mathsf{dup} \cdot q \equiv \mathsf{0}$,
- $q \cdot t \cdot \mathsf{dup} \cdot in \equiv \mathsf{0}$,
- q *drops w-tagged traffic,*

then

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

In a nutshell, this theorem says that executing the union of s and q is the same as sending packets through two separate copies of 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 theorem from Figure 4.

An interesting corollary of the result above is that when the ingress slice boundary of s and the domain of q do not overlap, for traffic destined for the ingress of s, the union of s and q is equivalent to 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*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- $w \neq w_0$,
- $out \cdot t \cdot \mathsf{dup} \cdot q \equiv \mathsf{0}$,
- $q \cdot t \cdot \mathsf{dup} \cdot in \equiv \mathsf{0}$,
- $in \cdot q \equiv \mathsf{0}$, *then*

$$\begin{array}{l} in \cdot \mathsf{tag} = w_0 \cdot ((s + q) \cdot t \cdot \mathsf{dup})^* \\ \equiv in \cdot \mathsf{tag} = w_0 \cdot (s \cdot t \cdot \mathsf{dup})^* \end{array}$$

Corollary 1 connects to traditional language-based information flow properties [27]. If s defines public, low-security data and q defines private, high security data, the corollary implies that the observable behavior of the network remains unchanged regardless of whether the high-security data (q) is present, absent, 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.

$$\begin{array}{lcl} in_1 & \triangleq & \mathsf{sw} = A \cdot \mathsf{pt} = 1 + \mathsf{sw} = B \cdot \mathsf{pt} = 2 \\ out_1 & \triangleq & \mathsf{sw} = A \cdot \mathsf{pt} = 1 + \mathsf{sw} = B \cdot \mathsf{pt} = 2 \\ s_1 & \triangleq & \{in_1\}\ w_1 : (p_{A1} + p_{B1})\ \{out_1\} \\ in_2 & \triangleq & \mathsf{sw} = A \cdot \mathsf{pt} = 3 + \mathsf{sw} = B \cdot \mathsf{pt} = 4 \\ out_2 & \triangleq & \mathsf{sw} = A \cdot \mathsf{pt} = 3 + \mathsf{sw} = B \cdot \mathsf{pt} = 4 \\ s_2 & \triangleq & \{in_2\}\ w_2 : (p_{A2} + p_{B2})\ \{out_2\} \end{array}$$

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

[3] In practice, the vlan field is often used to differentiate different classes of network traffic [34].

Pattern	Action	
typ=SSH	Drop	if typ=SSH then 0
port=1	Output 2	else if pt=1 then pt:=2
port=2	Output 1	else if pt=2 then pt:=1
*	Drop	else 0

Figure 9. A flow table and an equivalent NetKAT policy.

ONF Action Sequence	a	::=	$1 \mid f \leftarrow n \cdot a$
ONF Action Sum	as	::=	$0 \mid a + as$
ONF Predicate	b	::=	$1 \mid f = n \cdot b$
ONF Local	ℓ	::=	$as \mid \mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell$
ONF	p	::=	$0 \mid (\mathsf{sw} = sw \cdot \ell) + p$

Figure 10. OpenFlow Normal Form.

The original difficulty with this example was caused by packet duplication when, for example, a packet was sent from host 1 to host 2. Corollary 1 proves that we can use slices to 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.

7. Compilation

In order to execute a NetKAT program on an OpenFlow switch, we must compile it to a *flow table*, the low-level programming abstraction that OpenFlow supports. A flow table is a prioritized list of rules, where each rule consists of a pattern to match packet headers and actions to apply to matching packets. When a packet arrives at a switch, the actions associated with the highest priority matching rule are applied to it.

For example, the flow table on the left-hand side of figure 9 blocks SSH packets, but forwards all other traffic between ports 1 and 2. Alongside the flow table is an equivalent NetKAT policy. To make the connection between these two equivalent representations obvious, we introduce a conditional construct as shorthand:

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

Logically, a flow table pattern is a conjunction of positive literals, and each action is a combination of modifications. We can interpret prioritized rules as cascades of conditional expressions. In this section we describe the design of a compiler based on this idea.

The input to the compiler is NetKAT without dup or $\mathsf{sw} := n$ terms. These constructs are necessary to model network topology, but the output produced by the compiler is meant to execute on switches. Therefore, it is reasonable to exclude these features. The output of the compiler is a stylized subset of NetKAT called *OpenFlow Normal Form* (ONF). An ONF policy is a sum of conditional cascades, where each cascade is guarded by a test for a switch:

$$(\mathsf{sw} = sw_1 \cdot \ell_1) + \cdots + (\mathsf{sw} = sw_n \cdot \ell_n)$$

Each term can be interpreted as a complete flow table for a given switch. Figure 10 presents the full grammar for ONF. Mapping ONF to flow tables, is mostly straightforward, and many of the low-level details have been addressed in previous work [7].

The rest of this section outlines the major steps required to compile NetKAT to ONF. Each step eliminates or restricts an element of NetKAT syntax. In other words, each step translates from one intermediate representation to another until we arrive at ONF. We write $\text{NetKAT}^{-(op)}$ to denote NetKAT expressions that do not use the op operator. For example, if:

$$p \in \text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$$

then p does not contain dup and does not modify the sw field. This is the source language for the compiler, as described above.

Step 1: Star elimination. The first step is to eliminate Kleene star from the input policy. This step is critical as switches do not support iterated processing of packets—indeed, many switches only support a single phase of processing by a table! Formally, we prove that any program without dup, or, less importantly, assignment to sw, is equivalent to a Kleene star-free program (again without the dup primitive or assignments to sw).

Lemma 9 (Star Elimination)**.** *If $p \in \textit{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$, then there exists $p' \in \textit{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ where $p \equiv p'$.*

Proof. We show 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. By construction, dup can only appear in the normal form of an expression already containing dup, so p'' cannot contain dup. $R(p'') \subseteq I$ and p'' does not contain dup, so $R(p'') \subseteq At \cdot P$. Therefore, 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 $\mathsf{sw} = sw_i$. Therefore p'' also commutes with any test of the form $\mathsf{sw} = sw_i$. It follows that p'' can be written as a sum of $\alpha \cdot p$ for some atom α and complete assignment 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

$$\alpha \cdot (\mathsf{sw} = sw_i) \cdot p'' \cdot (\mathsf{sw} = sw_j) \geq \alpha \cdot p \neq 0$$
$$\alpha \cdot (\mathsf{sw} = sw_j) \cdot p'' \cdot (\mathsf{sw} = sw_i) = 0$$

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 $\mathsf{sw} = sw_i$. □

Step 2: Switch specialization. Next, we show that every star-free policy can be specialized for the switches in the network. This transformation allows us to remove nested tests of the sw field and put the policy into a form where it can easily be compiled into a flow table for each switch.

Lemma 10 (Switch Specialization)**.** *If $p \in \textit{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, then for all switches sw_i, there exists $p' \in \textit{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ such that $\mathsf{sw} = sw_i \cdot p \equiv \mathsf{sw} = sw_i \cdot p'$.*

Proof. Let g be the unique homomorphism of NetKAT defined on primitive programs by:

$$g(\mathsf{sw} = sw) \triangleq \begin{cases} 1 & \text{if } sw = sw_i \\ 0 & \text{otherwise} \end{cases}$$
$$g(f \leftarrow v) \triangleq f \leftarrow v$$
$$g(\mathsf{dup}) \triangleq \mathsf{dup}$$

For every primitive program element x of $\text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, we have both:

$$\mathsf{sw} = sw_i \cdot x \equiv g(x) \cdot \mathsf{sw} = sw_i$$
$$g(x) \cdot \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i \cdot g(x)$$

$$
\begin{array}{ll}
 & (\mathsf{if}\ \mathsf{dst} = A\ \mathsf{then}\ \mathsf{pt} := 1\ \mathsf{else}\ 0) + \\
 & (\mathsf{if}\ \mathsf{src} = B\ \mathsf{then}\ \mathsf{pt} := 2\ \mathsf{else}\ 0) \\
= & \mathsf{if}\ \mathsf{dst} = A \cdot \mathsf{src} = B\ \mathsf{then}\ \mathsf{pt} := 1 + \mathsf{pt} := 2 \\
 & \mathsf{else\ if}\ \mathsf{dst} = A\ \mathsf{then}\ \mathsf{pt} := 1 \\
 & \mathsf{else\ if}\ \mathsf{src} = B\ \mathsf{then}\ \mathsf{pt} := 2 \\
 & \mathsf{else}\ 0
\end{array}
$$

Figure 11. Compiling ONF policy union.

$$
\begin{array}{ll}
 & (\mathsf{if}\ \mathsf{typ} = \textsc{ssh}\ \mathsf{then}\ \mathsf{vlan} := W\ \mathsf{else}\ 1) \cdot \\
 & (\mathsf{if}\ \mathsf{dst} = A\ \mathsf{then}\ \mathsf{pt} := 1\ \mathsf{else\ if}\ \mathsf{dst} = B\ \mathsf{then}\ \mathsf{pt} := 2\ \mathsf{else}\ 0) \\
= & \mathsf{if}\ \mathsf{dst} = A \cdot \mathsf{typ} = \textsc{ssh}\ \mathsf{then}\ \mathsf{vlan} := W \cdot \mathsf{pt} := 1 \\
 & \mathsf{else\ if}\ \mathsf{dst} = A\ \mathsf{then}\ \mathsf{pt} := 1 \\
 & \mathsf{else\ if}\ \mathsf{dst} = B \cdot \mathsf{typ} = \textsc{ssh}\ \mathsf{then}\ \mathsf{vlan} := W \cdot \mathsf{pt} := 2 \\
 & \mathsf{else\ if}\ \mathsf{dst} = B\ \mathsf{then}\ \mathsf{pt} := 2 \\
 & \mathsf{else}\ 0
\end{array}
$$

Figure 12. Compiling ONF policy sequence.

Hence, applying KAT-COMMUTE [2, Lemma 4.4] twice shows:

$$\mathsf{sw} = sw_i \cdot p \equiv g(p) \cdot \mathsf{sw} = sw_i$$
$$g(p) \cdot \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i \cdot g(p)$$

By the definition of g, any occurrence of $\mathsf{sw} = v$ in p is replaced by either 1 or 0 in $g(p)$. Moreover, since $p \in \text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, it follows that $g(p)$ does not contain any occurrence of $\mathsf{sw} = v$ and since $p' = g(p) \in \text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ we also have

$$\mathsf{sw} = sw_i \cdot p \equiv \mathsf{sw} = sw_i \cdot p' \qquad \square$$

***Step 3: Converting to* ONF.** The third step is to compile policies in $\text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ to ONF. This is a recursive procedure that first compiles sub-policies to ONF.

To calculate the union of two ONF policies, we take the cross-product of the predicates and actions. This procedure is based on earlier work [23], but we present a purely syntactic proof of correctness. Figure 11 shows an example of compiling policy union, which illustrates why the cross-product construction is necessary.

Calculating the sequence of two ONF policies is more involved, since we have to commute the modifications in the first policy with the tests in the second policy to produce a single if-then-else cascade, as illustrated in figure 12. As the NetKAT axiom PA-MOD-FILTER-COMM shows, modifications and tests commute naively only if they affect distinct fields. The compiler has several transformations to ensure that various kinds of overlapping tests and modifications do commute.

Lemma 11 (Switch-local Compilation)**.**
If $p \in \text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ then there exists a policy p' such that $p \equiv p'$ and $p' \in$ ONF.

The proof goes by induction on the structure of p.

Step 4: Combining results. Lemmas 9, 10 and 11 suffice to prove any policy p in $\text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$ may be converted to OpenFlow normal form.

Theorem 7 (ONF)**.** *If $p_{in} \in \text{NetKAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$ then there exists $p_{out} \equiv p_{in}$ such that $p_{out} \in$ ONF.*

Optimizations. Naive compilation of network programs can produce flow tables that are unmanageably large [23]. Hence, existing systems implement optimizations to generate smaller tables. For example, the following lemma describes a common optimization called fall-through elimination, which removes unnecessary rules from the table.

Lemma 12 (Fall-through Elimination)**.** *If $b_1 \leq b_2$ then*

$$\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else\ if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell\ \equiv \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$$

We plan to study further optimizations in future work.

8. Related Work

Kleene algebra is named for its inventor, Stephen Cole Kleene. Much of the basic algebraic theory of KA was developed by John Horton Conway [4]. Kleene algebra with tests was introduced by Kozen [13, 14]. KA and KAT have been successfully applied in many practical verification tasks, including verification of compiler optimizations [16], pointer analysis [22], concurrency control [3], and device drivers [15]. This is the first time KA has been used as a network programming language or applied to verification of networks. The proof of the main result in this paper—completeness of the equational axioms—is based on a novel model of KAT.

While many other systems have been proposed for analyzing networks, we believe ours is the first to provide a complete, high-level algebra for reasoning about network programs *as they are written*. Systems such as Anteater [19], FlowChecker [1], Header Space Analysis [10], VeriFlow [11], and Formally Verifiable Networking [33], encode information about network topology and forwarding policies into SAT formulae (Anteater), graph-based representations (VeriFlow, Header Space Analysis), or higher-order logic (Formally Verifiable Networking). These systems then define custom algorithms over these models to check specific properties such as reachability or packet loss. Such systems can check for violations of important network invariants, but do not provide sound and complete systems for reasoning *directly* about programs. Moreover, although these systems have expressive languages for encoding properties, they do not connect these encodings back to denotational or operational models of the network. In contrast, in section 5, we show how to encode a reachability property as a NetKAT equation and then prove that the reachability equation is equivalent to a semantic definition of reachability.

As a programming language, NetKAT is most similar to NetCore [7, 23] and Pyretic [24], which both stemmed from earlier work on Frenetic [6]. NetCore defined the fragment of NetKAT with filters, modification and union, and Pyretic extended NetCore with sequential composition (although Pyretic has neither a formal semantics nor a compiler). Neither language defined an equational theory for reasoning about programs, nor did they include Kleene star—unlike these previous languages, NetKAT programs can describe potentially infinite behaviors.

NDLog [18] 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 [29]. NetKAT's Kleene star is able to model network behavior, but has decidable (PSPACE-complete) equivalence.

9. Conclusion

This paper presents NetKAT, a new language for SDN programming and reasoning that is based on a solid semantic foundation—Kleene algebra with tests. NetKAT's denotational semantics describes network programs as functions from packet histories to sets of packets histories and its equational theory is sound and complete with respect to this model. The language enables programmers to create expressive, compositional network programs and reason effectively about their semantics. We demonstrate the power of our framework on a range of practical applications including reachability, traffic isolation, access control, and compiler correctness.

Acknowledgments The authors wish to thank Timothy Griffin, Shriram Krishnamurthi, Nick McKeown, Jennifer Rexford, the members of the Cornell PLDG, and the POPL reviewers for helpful comments, as well as Alec Story and Stephen Gutz for work on a preliminary version of slices. This work is supported in part by the NSF under grants CNS-1111698, CNS-0931985, CNS-1111520, and SHF-1016937, the ONR under award N00014-12-1-0757, a Sloan Research Fellowship, and a Google Research Award.

References

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

[3] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J., 1994.

[4] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[5] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[6] 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.

[7] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, June 2013.

[8] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[9] James Hamilton. Networking: The last bastion of mainframe computing, December 2009. Available at `http://tinyurl.com/y9uz64e`.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[12] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *I&C*, 110(2):366–390, May 1994.

[13] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *TACAS*, pages 14–33, Passau, Germany, March 1996.

[14] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[15] Dexter Kozen. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Computer Science Department, Cornell University, November 2003.

[16] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In *CL*, July 2000.

[17] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, September 1996.

[18] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.

[19] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[20] James McCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. Extending SDN to large-scale networks. In *ONS*, 2013.

[21] 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.

[22] B. Möller. Calculating with pointer structures. In *Algorithmic Languages and Calculi. Proc. IFIP TC2/WG2.1 Working Conference*, February 1997.

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

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

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

[26] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[27] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[28] Gunther Schmidt. *Relational Mathematics*. Cambridge University Press, 2010.

[29] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *PODS*, pages 237–249, 1987.

[30] The Frenetic Project, 2013. See `http://frenetic-lang.org`.

[31] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

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

[33] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally veriable networking. In *HotNets*, 2009.

[34] 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.