A Coalgebraic Decision Procedure for NetKAT

Nate Foster
Cornell University *

Dexter Kozen
Cornell University*

Mae Milano
Cornell University*

Alexandra Silva
Radboud University Nijmegen †

Laure Thompson
Cornell University*

Abstract

Program equivalence is a fundamental problem that has practical applications across a variety of areas of computing including compilation, optimization, software synthesis, formal verification, and many others. Equivalence is undecidable in general, but in certain settings it is possible to develop domain-specific languages that are expressive enough to be practical and yet sufficiently restricted so that equivalence remains decidable.

In previous work we introduced NetKAT, a domain-specific language for specifying and verifying network packet-processing functions. NetKAT provides familiar constructs such as tests, assignments, union, sequential composition, and iteration as well as custom primitives for modifying packet headers and encoding network topologies. Semantically, NetKAT is based on Kleene algebra with tests (KAT) and comes equipped with a sound and complete equational theory. Although NetKAT equivalence is decidable, the best known algorithm is hardly practical—it uses Savitch's theorem to determinize a PSPACE algorithm and requires quadratic space.

This paper presents a new algorithm for deciding NetKAT equivalence. This algorithm is based on finding bisimulations between finite automata constructed from NetKAT programs. We investigate the coalgebraic theory of NetKAT, generalize the notion of Brzozowski derivatives to NetKAT, develop efficient representations of NetKAT automata in terms of spines and sparse matrices, and discuss the highlights of our prototype implementation.

1. Introduction

Determining whether a given pair of programs is equivalent is a fundamental question that arises in many areas of computing. For instance, to verify that a complicated program satisfies a specification one can often encode the specification as a simple program and check that the implementation is equivalent to the encoded specification. To establish the correctness of a compiler, one can check that the output produced by the compiler is semantically equivalent to the program provided as input. Equivalence is undecidable in general, but in certain settings it is possible to develop domain-specific languages that are expressive enough to be practical and yet sufficiently restricted so that equivalence remains decidable.

* Department of Computer Science, Cornell University, Ithaca, NY, USA

† Institute for Computing and Information Sciences, Radboud University Nijmegen, 6525 AJ Nijmegen, The Netherlands. Also affiliated to Centrum Wiskunde & Informatica (Amsterdam, The Netherlands) and HASLab/INESC TEC, Universidade do Minho (Braga, Portugal). Work performed at Cornell University.

Networks have received a lot of attention in recent years as a potential target for domain-specific language design, due to the recent emergence of software-defined networking (SDN) as an open and flexible platform for network programming. SDN is now being deployed in production enterprise, data center, and wide-area networks [15, 18, 19]. Meanwhile, researchers in the programming languages community have developed a number of SDN languages including Frenetic, Pyretic, Nettle, Maple, PANE, and others [11–13, 25, 26, 35, 36]. The details of these languages differ, but each aims to provide high-level abstractions that simplify the task of specifying packet-processing programs. To help test and debug such programs, verification tools such as HSA, VeriFlow, FlowLog, and VeriCon are also being actively developed [2, 16, 17, 27]. Given the significant interest in applications, languages, and verification tools for SDN, we believe that having mechanisms for deciding when a given pair of programs are equivalent would provide substantial value.

NetKAT Language. In previous work we introduced NetKAT, a domain-specific language for specifying and verifying packet-processing functions in a network [1]. NetKAT is based on Kleene algebra with tests (KAT), a mathematical framework that combines Kleene algebra (KA), the algebra of regular expressions, with Boolean algebra. The language provides familiar constructs such as tests, assignments, union, sequential composition, and iteration as well as special-purpose primitives for modifying packet headers and encoding network topologies.

The original NetKAT design defined a rigorous semantic and axiomatic basis for network programming including: (i) a denotational semantics consisting of a model based on packet-processing functions as well as an equivalent language model; (ii) a sound and complete quasi-equational axiomatization; and (iii) a proof that the equational theory of NetKAT is PSPACE-complete. In addition, we developed practical applications of the equational theory and showed how it could be used to address issues such as reasoning about reachability, acyclicity, isolation, and compiler correctness.

Unfortunately the algorithm for deciding equivalence proposed in our initial work on NetKAT is hardly practical—it uses Savitch's theorem to convert a nondeterministic PSPACE algorithm to deterministic PSPACE. Although it suffices to establish the complexity bound, the algorithm is undesirable as a basis for implementation because it is likely exponential time in both the best and worst case, and it involves a quadratic blowup in space. However, recent experiences implementing KA and KAT [4, 5, 28] suggest that the worst-case complexity may not be an impediment in common cases, and that the system may be amenable to automation provided that the problem is approached correctly.

Coalgebraic Techniques. This paper brings coalgebraic techniques to bear on the problem of deciding NetKAT equivalence [4, 5, 28, 30, 32]. Coalgebra is a framework for modeling and reasoning about state-based systems that provides general techniques for formalizing and reasoning about correctness. It allows a uniform and systematic development of classical constructions which have previously been derived in an ad hoc fashion. Working in terms of coalgebra also facilitates exploiting dualities to obtain canonical constructions of an appropriate logic or algebra from a combinatorial or coalgebraic state-based model (or vice versa). These constructions are often accompanied by substantial simplifications or generalizations compared to ad hoc techniques.

A key aspect of the coalgebraic approach is that the canonical notion of equivalence is derived from the type of the system. Consider deterministic finite automata (DFA), which consist of a set of states S, together with a function $o\colon S \to 2$ that determines final states, and a state transition function $t\colon S \to S^\Sigma$. One can think of a DFA as the pair $(S, \langle o, t\rangle\colon S \to 2 \times S^\Sigma)$, which is a coalgebra for the functor $F(S) = 2 \times S^\Sigma$. Moreover, this type is rich enough to derive canonical notions of behavior and equivalence. In the case of the automaton type these are formal languages and language equivalence respectively.

More generally, in coalgebraic models the canonical universe of behaviors has a coalgebraic structure for the same type. For example, on sets of strings this structure is given by the Brzozowski derivative consisting of *continuation* and *observation maps*

$$D\colon 2^{\Sigma^*} \to (2^{\Sigma^*})^A \qquad E\colon 2^{\Sigma^*} \to 2$$

$$D_a(A) = \{x \mid a \cdot x \in A\} \qquad E(A) = \begin{cases} 1, & \text{if } \varepsilon \in A, \\ 0, & \text{if } \varepsilon \notin A, \end{cases}$$

where ε is the null string [7]. The Brzozowski derivative can also be defined syntactically on regular expressions by structural induction.

Importantly, this structure has the property of being final for the automaton type—that is, any other coalgebra of the same type has a unique mapping into it. In the case of an automaton, this map assigns each state to the language it accepts. The interpretation of a regular expression as a regular set of strings is the unique coalgebra morphism from the coalgebra of regular expressions to the coalgebra of sets of strings. The image of an expression e and its derivatives under this map is the minimal automaton for the set represented by e. Deciding equivalence of regular expressions is tantamount to deciding equivalence of the coalgebras of derivatives of the two expressions. The situation for KAT is similar, except that the strings over a finite set of characters Σ are replaced by *guarded strings* over primitive actions Σ and tests T [22]. The coalgebraic approach has been successfully used to decide equivalence for KA and KAT in previous work by Pous and others [4, 5, 28].

The essential step in all of these results is to identify a suitable *language model* that characterizes the equational theory of the system at hand. A language model is a family of structures in which expressions are interpreted as sets of elements of a particular monoid, where the language models form the free models of the system itself. An additional benefit of having a language model is that it is typically a key component of the coalgebraic theory, forming the final coalgebra for an appropriate signature defined by a specialized version of the Brzozowski derivative.

In the cases of KA and KAT, the language models are the regular sets of strings over Σ and the regular sets of guarded strings over Σ and T, respectively. The situation is the same with NetKAT: in our earlier work, we identified a language model that is isomorphic to the standard semantics based on packet-processing functions [1]. However, as NetKAT incorporates a number of extra equational premises as part of the theory, the appropriate language model is correspondingly more involved, consisting of the regular sets of strings of a certain reduced form.

Contributions. This paper presents a new technique for deciding the equivalence of NetKAT programs, following the coalgebraic approach just described. Its main contributions are as follows:

- We formalize the coalgebraic theory of NetKAT, including a semantic version presented in terms of regular sets of a certain restricted class of strings and a syntactic version in terms of NetKAT expressions.
- We present a surprising characterization of Brzozowski derivatives in terms of sparse matrices over spines calculated inductively from NetKAT expressions.
- We use these insights to develop an efficient implementation of our algorithm in OCaml that often avoids the worst case PSPACE time and quadratic space of earlier algorithms.

Overall, these results provide a solid theoretical basis and initial steps toward a practical implementation that will make it possible to decide equivalence in a wide variety of applications.

Outline. The rest of this paper is organized as follows. §2 reviews the definitions of KAT and NetKAT from [1] and gives an overview of the main technical tools used in the rest of this paper, including the Brzozowski derivative. §3 introduces NetKAT coalgebras and automata and establishes a number of technical lemmas that are needed for our main results. This section includes a definition of a variant of the Brzozowski derivative for NetKAT with syntactic and semantic versions, and also describes our matrix representation. §4 proves our main theoretical result on which the correctness of our algorithm is based: a version of Kleene's theorem for NetKAT, which states that NetKAT expressions and NetKAT automata are equally expressive. §5 presents highlights of our implementation of the decision algorithm and explains how the coalgebraic approach gives improved performance over the naive algorithm of [1]. §6 discusses related work. Finally, §7 presents conclusions and directions for future work.

2. Overview

This section briefly reviews the syntax and semantics of NetKAT [1], as well as classic work on derivatives of regular expressions [7], to set the stage for the results described in subsequent sections.

NetKAT Language. NetKAT is a framework for programming and reasoning about networks [1]. When used for programming, NetKAT can be understood in terms of a denotational semantics that models the meaning of each program as a function from packet histories to sets of packet histories (where a packet history is a non-empty sequence of packets). When used for verification, NetKAT can be understood in terms of a set of first-order axioms that capture equivalences between programs. The language provides constructs for encoding network topologies, primitives for filtering and modifying packets, and a rich collection of composition operators that combine programs using the union, sequential composition, and iteration operators. The NetKAT compiler and run-time system translates programs written in this language into low-level instructions that can be installed on switches that process packets efficiently in hardware.

More formally, we model a *packet* as a record with a fixed set of fields $f_1, \ldots, f_k$, each mapping to constants that represent various types of packet data such as Ethernet addresses, VLAN tags, IP addresses, protocol types, TCP ports, etc. To support reasoning about NetKAT programs, we model a *packet history* as a non-empty sequence of packets. Intuitively, a history captures the trajectory of a single packet as it traverses the physical topology.

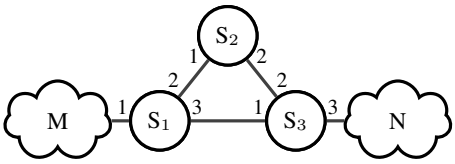


Figure 1. Example topology.

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*
	$\mid$	$\overline{a}$	*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

$$\begin{aligned}
[\![p]\!] &\in \mathsf{H} \to 2^{\mathsf{H}} \\
[\![\mathsf{1}]\!]\ h &\triangleq \{h\} \\
[\![\mathsf{0}]\!]\ h &\triangleq \varnothing \\
[\![f = n]\!]\ (pk{::}h) &\triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \varnothing & \text{otherwise} \end{cases} \\
[\![\overline{a}]\!]\ h &\triangleq \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ (pk{::}h) &\triangleq \{pk[f := n]{::}h\} \\
[\![p + q]\!]\ h &\triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cdot q]\!]\ h &\triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p^*]\!]\ h &\triangleq \textstyle\bigcup_{i \in \mathbb{N}} F^i\ h \\
\text{where } F^0\ h &\triangleq \{h\} \text{ and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\mathsf{dup}]\!]\ (pk{::}h) &\triangleq \{pk{::}(pk{::}h)\}
\end{aligned}$$

Figure 2. NetKAT: syntax and semantics.

At the level of syntax, NetKAT is divided into two categories: predicates $a, b, \ldots$ and policies $p, q, \ldots$. Predicates include the constants true (1) and false (0) and tests ($f = n$), and Boolean negation (!), conjunction ($\cdot$), and disjunction ($+$), while policies include predicates, primitive assignments $f \leftarrow n$, and duplication dup, and are closed under union ($+$), sequential composition ($\cdot$), and iteration (*). Semantically, every NetKAT expression p denotes a function $[\![p]\!] : \mathsf{H} \to 2^{\mathsf{H}}$ that takes a packet history h and generates a (possibly empty) set of histories $\{h_1, \ldots, h_n\}$. Generating the empty set corresponds to dropping the packet, a singleton corresponds to modifying or forwarding it, and larger sets correspond to duplicating the packet and broadcasting it to several locations. The syntax and semantics of NetKAT is defined formally in Figure 2.

Example. As an example to illustrate the key features of NetKAT, consider the topology shown in Figure 1. It consists of three switches (S1, S2, and S3) arranged in a triangle that provide connectivity between two other networks (M and N). The annotation on the endpoint of each link indicates the physical port on the adjacent switch it is connected to.

We can encode the topology as a simple NetKAT program:

$$\begin{aligned}
t \triangleq\ & (\mathsf{sw} = \mathrm{S}_1 \cdot \mathsf{pt} = 2 \cdot \mathsf{sw} \leftarrow \mathrm{S}_2 \cdot \mathsf{pt} \leftarrow 1) + \\
& (\mathsf{sw} = \mathrm{S}_1 \cdot \mathsf{pt} = 3 \cdot \mathsf{sw} \leftarrow \mathrm{S}_3 \cdot \mathsf{pt} \leftarrow 1) + \\
& (\mathsf{sw} = \mathrm{S}_2 \cdot \mathsf{pt} = 1 \cdot \mathsf{sw} \leftarrow \mathrm{S}_1 \cdot \mathsf{pt} \leftarrow 2) + \\
& (\mathsf{sw} = \mathrm{S}_2 \cdot \mathsf{pt} = 2 \cdot \mathsf{sw} \leftarrow \mathrm{S}_3 \cdot \mathsf{pt} \leftarrow 2) + \\
& (\mathsf{sw} = \mathrm{S}_3 \cdot \mathsf{pt} = 1 \cdot \mathsf{sw} \leftarrow \mathrm{S}_1 \cdot \mathsf{pt} \leftarrow 3) + \\
& (\mathsf{sw} = \mathrm{S}_3 \cdot \mathsf{pt} = 2 \cdot \mathsf{sw} \leftarrow \mathrm{S}_2 \cdot \mathsf{pt} \leftarrow 2)
\end{aligned}$$

The top-level program is the union ($+$) of several smaller programs, one for each link in the topology. The link programs are sequential compositions ($\cdot$) of filters and modifications where the filters discard packets not located at the link ingress (as identified by a switch and port), and the modifications relocate packets to link egress.

We can also describe the packet-processing behavior of the switches themselves in NetKAT program. For example, the following program forwards traffic from M to N:

$$\begin{aligned}
p \triangleq\ & (\mathsf{sw} = \mathrm{S}_1 \cdot \mathsf{pt} = 1; \\
& \quad (\mathsf{typ} = \mathrm{HTTP} \cdot \mathsf{pt} \leftarrow 2 + \overline{\mathsf{typ} = \mathrm{HTTP}} \cdot \mathsf{pt} \leftarrow 3) + \\
& (\mathsf{sw} = \mathrm{S}_2 \cdot \mathsf{pt} = 1 \cdot \mathsf{pt} \leftarrow 2) + \\
& (\mathsf{sw} = \mathrm{S}_3 \cdot (\mathsf{pt} = 1 + \mathsf{pt} = 2) \cdot \mathsf{pt} \leftarrow 3)
\end{aligned}$$

It sends HTTP traffic on the two-hop path from S_1 to S_2 to S_3, and all other traffic on the direct path from S_1 to S_3.

To obtain a complete program, we can sequentially compose the switch program p with the topology program t and iterate the result to obtain a program $(p \cdot t)^*$ that interleaves arbitrarily many steps of processing on switches and using the topology.

Formal Reasoning. NetKAT can also be used as a formal reasoning system—for example, we can use it to establish that the program above correctly forwards all traffic from M to N. To do this, we can write the simplest program that meets this specification,

$$s \triangleq (\mathsf{sw} = \mathrm{S}_1 \cdot \mathsf{pt} = 1 \cdot \mathsf{sw} \leftarrow \mathrm{S}_3 \cdot \mathsf{pt} \leftarrow 3)$$

and prove that it is equivalent to the actual implementation:

$$s \equiv (p \cdot t)^*$$

To support formal reasoning about equivalences between programs, NetKAT is carefully designed to be an instance of a Kleene algebra with tests (KAT) [21]. A *Kleene algebra* (KA) is an algebraic structure $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ where $(K, +, \cdot, 0, 1)$ is an idempotent semiring and p^*q (respectively qp^*) is the least solution of the affine linear inequality $q + pr \leq r$ (respectively $q + rp \leq r$), where $p \leq q$ is an abbreviation for $p + q = q$. A *Kleene algebra with tests* (KAT) is a two-sorted algebraic structure,

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

where $^-$ is a unary operator defined only on B, such that

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

In the literature on KAT, the elements of B and K are usually called *tests* and *actions* respectively. In the rest of this paper, we will use the terms predicate/test and policy/action interchangeably, and will often elide the $\cdot$ operator in examples. In addition to the standard axioms, NetKAT satisfies some extra axioms governing the packet-processing primitives that that are not valid for KATs in general. The complete set of axioms for NetKAT is given in Figure 3.

Reduced NetKAT. The standard model of NetKAT just defined is based on functions from packet histories to sets of packet histories.

Kleene Algebra Axioms

$p + (q + r) \equiv (p + q) + r$	KA-PLUS-ASSOC
$p + q \equiv q + p$	KA-PLUS-COMM
$p + 0 \equiv p$	KA-PLUS-ZERO
$p + p \equiv p$	KA-PLUS-IDEM
$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$	KA-SEQ-ASSOC
$1 \cdot p \equiv p$	KA-ONE-SEQ
$p \cdot 1 \equiv p$	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
$0 \cdot p \equiv 0$	KA-ZERO-SEQ
$p \cdot 0 \equiv 0$	KA-SEQ-ZERO
$1 + p \cdot p^* \equiv p^*$	KA-UNROLL-L
$q + p \cdot r \leq r \Rightarrow p^* \cdot q \leq r$	KA-LFP-L
$1 + p^* \cdot p \equiv p^*$	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
$a + 1 \equiv 1$	BA-PLUS-ONE
$a + \neg a \equiv 1$	BA-EXCL-MID
$a \cdot b \equiv b \cdot a$	BA-SEQ-COMM
$a \cdot \neg a \equiv 0$	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
$f = n \cdot f = n' \equiv 0$, if $n \neq n'$	PA-CONTRA
$\sum_i f = i \equiv 1$	PA-MATCH-ALL

Figure 3. NetKAT: equational axioms.

This is a convenient for programming and manual proofs, but it is less amenable to automation. As an intermediate step toward an efficient procedure for deciding NetKAT equivalence, we next define a simpler variant of NetKAT called *reduced NetKAT* in which tests and assignments specify the value of every packet field. Let $f_1, \ldots, f_k$ be a list of all fields of a packet in some fixed order. For each tuple $\overline{n} = n_1, \ldots, n_k$ of values, let $\overline{f} = \overline{n}$ and $\overline{f} \leftarrow \overline{n}$ denote the expressions

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

called *complete tests* and *complete assignments* respectively. Complete tests are also called atoms because they are the atomic (minimal nonzero) elements of the Boolean algebra generated by B. Complete tests and assignments are in one-to-one correspondence according to the values $\overline{n}$. The complete assignment corresponding to the atom α is denoted p_α, and the atom corresponding to complete assignment p is denoted α_p. The sets of atoms and complete assignments are denoted by At and P, respectively.

The NetKAT axioms entail the following properties of atoms and complete assignments:

$$p = p\alpha_p \qquad \alpha\,\mathsf{dup} = \mathsf{dup}\,\alpha \qquad \sum_\alpha \alpha = 1$$

$$\alpha = \alpha p_\alpha \qquad pp' = p' \qquad \alpha\beta = 0,\ \alpha \neq \beta.$$

Every policy is provably equivalent to a policy in which all primitive assignments $f \leftarrow n$ appear in the context of a complete assignment, and every test is equivalent to a sum of atoms. For this reason, we can view any NetKAT policy as a KAT expression over the alphabet $P \cup \mathsf{At} \cup \{\mathsf{dup}\}$.

Language Model. As shown in [1], NetKAT has a natural language model that plays the same role as that played by ordinary strings in KA and guarded strings in KAT. These language models are important because they are free algebras for their respective algebraic systems. That is, they capture the equational theory of an entire class of algebras in a single concrete interpretation. As with KA and KAT, the language model for NetKAT is defined as the family of regular sets over a typed monoid. A typed monoid is simply a small category where the types are the objects and the elements of the monoid are the morphisms. The types are used to exclude certain products: the product xy exists if and only if the codomain of x matches the domain of y. Instead of a single identity element, a typed monoid has a component 1_α for every object α. The regular sets over a typed monoid M are the smallest family of subsets of M containing the singletons and closed under the KAT operations $+$, $\cdot$, *, $^-$, 1, and 0, where

$$A + B = A \cup B \quad AB = \{xy \mid x \in A,\ y \in B,\ xy \text{ exists}\}$$

$$A^* = \textstyle\bigcup_n A^n, \text{ where } A^0 = 1 \text{ and } A^{n+1} = A \cdot A^n$$

$$\overline{A} = 1 - A,\ A \subseteq 1 \quad 1 = \{1_\alpha \mid \alpha \text{ is an object}\} \quad 0 = \varnothing.$$

It is straightforward to show that the regular sets over M form a KAT.

The language model for KA is the family of regular sets over the free monoid Σ^*, where Σ is a finite alphabet. The category has only one object, and all products xy exist. The element 1 is the singleton $\{\varepsilon\}$, where ε is the null string. This is a KAT, although the Boolean component is the trivial two-element Boolean algebra. There is a standard interpretation R from regular expressions to regular sets defined as the unique homomorphism such that $R(a) = \{a\}$ for $a \in \Sigma$, $R(1) = \{\varepsilon\}$, and $R(a) = 1$, and $R(e_1) = R(e_2)$ iff e_1 and e_2 represent the same element in all Kleene algebras [20].

The language model for KAT is the family of regular sets of guarded strings over a set of primitive tests B and a set of primitive actions Σ. A guarded string is a sequence

$$\alpha_0 p_1 \alpha_1 p_2 \alpha_2 \cdots p_{n-1} \alpha_{n-1} p_n \alpha_n,\ n \geq 0,$$

where the α_i are atoms of the free Boolean algebra on generators B and the p_i are elements of Σ. The objects of the category are the atoms α, and the guarded string $\alpha x \beta$ is a morphism of type $\alpha \to \beta$. The components of the identity are guarded strings of length 0, which are just the atoms α. Multiplication on guarded strings is the fusion product:

$$x\alpha \cdot \beta y = \begin{cases} x\alpha y & \text{if } \alpha = \beta, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

As above, there is a standard interpretation G that maps KAT expressions to regular sets of guarded strings defined as the unique homomorphism such that $G(p) = \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\}$ for $p \in \Sigma$ and $G(b) = \{\alpha \mid \alpha \leq b\}$ for $b \in B$. Again, e_1 and e_2 agree under G iff they agree under all interpretations [23].

For NetKAT, we have a similar language model formed from a typed monoid. The monoid consists of strings in the set

$$U = \mathsf{At} \cdot (P \cdot \mathsf{dup})^* \cdot P,$$

where P and At range over complete tests and assignments respectively; that is, strings of the form

$$\alpha p_0\ \mathsf{dup}\ p_1\ \mathsf{dup} \cdots \mathsf{dup}\ p_n,\ n \geq 0,$$

where $p_i \in P$ and $\alpha \in \mathsf{At}$. The types are the atoms At, and the string $\alpha x p$ is of type $\alpha \to \alpha_p$. Multiplication is thus

$$xp \cdot \alpha y = \begin{cases} xy & \text{if } \alpha = \alpha_p, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The components of the identity are the strings αp_α.

As above, the regular sets over this typed monoid form a KAT, and there is an interpretation G from KAT expressions over primitive tests B and primitive actions $P \cup \{\mathsf{dup}\}$ to this KAT, namely the unique homomorphism such that

$$G(p) = \{\alpha p \mid \alpha \in \mathsf{At}\}$$
$$G(b) = \{\alpha p_\alpha \mid \alpha \leq b\}$$
$$G(\mathsf{dup}) = \{\alpha p_\alpha \ \mathsf{dup}\ p_\alpha \mid \alpha \in \mathsf{At}\}.$$

As shown in [1], $G(e_1) = G(e_2)$ iff $[\![e_1]\!] = [\![e_2]\!]$, where $[\![e]\!]$ is the standard interpretation of NetKAT expressions as packet-forwarding programs in a network, and the axioms of NetKAT are complete for the equational theory of these interpretations.

3. NetKAT Coalgebra and Automata

There is a coalgebraic theory of NetKAT that provides a combinatorial view of reduced NetKAT in much the same way that classical automata theory does for KA and automata on guarded strings do for KAT.

3.1 Definitions

A NetKAT coalgebra consists of a set of states S along with *continuation* and *observation maps*

$$\delta_{\alpha\beta} : S \to S \qquad \varepsilon_{\alpha\beta} : S \to 2$$

for $\alpha, \beta \in \mathsf{At}$. The continuation and observation maps play the role of transitions and final states in ordinary automata theory. A deterministic NetKAT automaton is simply a finite-state NetKAT coalgebra with a distinguished start state $s \in S$. There is also a notion of nondeterministic automaton and a determinization procedure, but we will not need this for our development.

Inputs are strings in $U = \mathsf{At} \cdot P \cdot (\mathsf{dup} \cdot P)^*$, that is, strings of the form

$$\alpha p_0 \ \mathsf{dup}\ p_1 \ \mathsf{dup} \cdots \mathsf{dup}\ p_n$$

for some $n \geq 0$. Intuitively, $\delta_{\alpha\beta}$ attempts to consume $\alpha p_\beta \ \mathsf{dup}$ from the front of the input string and move to a new state with a residual input string. This will succeed iff the string is of the form $\alpha p_\beta \ \mathsf{dup}\ x$ for some $x \in (P \cdot \mathsf{dup})^*$, in which case the automaton moves to a new state as determined by $\delta_{\alpha\beta}$ with residual input string βx. The observation map $\varepsilon_{\alpha\beta}$ determines whether the string is αp_β should be accepted in the current state.

Formally, acceptance is determined by a coinductively defined predicate $\mathsf{Accept} : S \times U \to 2$:

$$\mathsf{Accept}(t, \alpha p_\beta \ \mathsf{dup}\ x) = \mathsf{Accept}(\delta_{\alpha\beta}(t), \beta x)$$
$$\mathsf{Accept}(t, \alpha p_\beta) = \varepsilon_{\alpha\beta}(t).$$

A string $x \in U$ is *accepted* by the automaton if $\mathsf{Accept}(s, x)$, where s is the start state.

Thus a NetKAT coalgebra is a coalgebra for the set endofunctor

$$FX = X^{\mathsf{At}\times\mathsf{At}} \times 2^{\mathsf{At}\times\mathsf{At}} \tag{3.1}$$

The continuation and observation maps comprise the structure map of the coalgebra:

$$(\delta, \varepsilon) : X \to FX.$$

One can see immediately from (3.1) that $X^{\mathsf{At}\times\mathsf{At}}$ and $2^{\mathsf{At}\times\mathsf{At}}$ are isomorphic to the families of square matrices over X and 2, respectively, with rows and columns indexed by At. Indeed, we exploited the one-to-one correspondence between P and At to write δ and ε in this form. This simple observation will turn out to play a key role in our subsequent development.

3.2 The Brzozowski Derivative

In this section we describe a variant of the Brzozowski derivative for NetKAT. The derivative comes in two versions: semantic and syntactic. The semantic version is defined on subsets of U and gives rise to a NetKAT coalgebra $(2^U, \delta, \varepsilon)$, which is final for the NetKAT signature. The syntactic version is defined on NetKAT expressions and alse gives rise to a coalgebra (Exp, D, E). The standard interpretation $G : \mathsf{Exp} \to 2^U$ is the unique coalgebra morphism to the final coalgebra.

Brzozowski Derivatives. As review, Brozozowski derivatives are a simple technique for constructing a deterministic finite automaton from a regular expression. We briefly review the definition of the Brzozowski derivative to set the stage for the generalized versions presented in following sections. Generally speaking, given an alphabet Σ and a regular language L, the (left-) derivative of a language $L \subseteq \Sigma^*$ with respect to a symbol a in Σ is the set of strings x such that the string ax is in L.

The derivative can also be defined inductively on the structure of regular expressions. It uses an auxiliary function $\varepsilon(e)$, which produces 1 if the empty string is an element of the language denoted by e and 0 otherwise.

$$\delta_a(e_1 + e_2) = \delta_a(e_1) + \delta_a(e_2) \qquad \delta_a(e^*) = \delta_a(e) \cdot e^*$$
$$\delta_a(e_1 e_2) = \delta_a(e_1) \cdot e_2 + \varepsilon(e_1) \cdot \delta_a(e_2)$$
$$\delta_a(b) = [a = b] \qquad \delta_a(0) = 0 \qquad \delta_a(1) = 0$$

where for a predicate φ,

$$[\varphi] = \begin{cases} 1, & \text{if } \varphi \text{ is true,} \\ 0, & \text{if } \varphi \text{ is false.} \end{cases}$$

Likewise, the ε function has a simple inductive definition:

$$\varepsilon(e_1 + e_2) = \varepsilon(e_1) + \varepsilon(e_2) \qquad \varepsilon(e_1 e_2) = \varepsilon(e_1) \cdot \varepsilon(e_2)$$
$$\varepsilon(e^*) = 1 \qquad \varepsilon(a) = 0 \qquad \varepsilon(0) = 0 \qquad \varepsilon(1) = 1$$

To construct an automaton from an expression e, we take the set of states to be the set of regular expressions obtained by taking repeated derivatives from e modulo associativity and commutativity of $\cdot$, associativity, commutativity, and idempotence of $+$. We let e be the initial state, and we mark a state as final if ε produces 1 when applied to that state. Finally, we include transition from e_i to e_j on a if $\delta_a(e_i) = e_j$. Given the automaton representation of an expression, many algorithms become easy to implement, including testing for equivalence. One of the main contributions of this paper will be to develop analogous constructions for NetKAT.

Final Coalgebra. The classical set-theoretic Brzozowski derivative for KA consisting of continuation and observation maps

$$\delta_a : 2^{\Sigma^*} \to 2^{\Sigma^*} \qquad \varepsilon : 2^{\Sigma^*} \to 2$$
$$\delta_a(A) = \{x \in \Sigma^* \mid ax \in A\} \qquad \varepsilon(A) = [\varepsilon \in A]$$

constitute the final coalgebra for the functor $FX = X^\Sigma \times 2$ relevant to the study of KA and classical automata theory [30, 32]. There is also a version for KAT and automata on guarded strings:

$$\delta_{\alpha p} : 2^{GS} \to 2^{GS} \qquad \varepsilon_\alpha : 2^{GS} \to 2$$
$$\delta_{\alpha p}(A) = \{x \in GS \mid \alpha p x \in A\} \qquad \varepsilon_\alpha(A) = [\alpha \in A],$$

where GS is the set of guarded strings [22].

For NetKAT, we have a similar construction:

$$\delta_{\alpha\beta} : 2^U \to 2^U \qquad \varepsilon_{\alpha\beta} : 2^U \to 2$$
$$\delta_{\alpha\beta}(A) = \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in A\} \quad \varepsilon_{\alpha\beta}(A) = [\alpha p_\beta \in A].$$

One can show that this is the final coalgebra for the NetKAT signature by showing that bisimilarity implies equality, but we will not need this fact for our development.

Syntactic Coalgebra. In addition to the set-theoretic Brzozowski derivative for KA, there is a syntactic version consisting of continuation and observation maps

$$D_a : \mathsf{Exp} \to \mathsf{Exp} \qquad E : \mathsf{Exp} \to 2$$

where Exp is the set of regular expressions over a finite alphabet Σ. The syntactic Brzozowski derivative operates on regular expressions and can be defined by induction on the structure of the expression. The set of all derivatives of e is finite modulo ACI (associativity, commutativity, and idempotence of +) and gives a deterministic finite automaton accepting the set of strings in Σ^* represented by e under the canonical interpretation $R : \mathsf{Exp} \to 2^{\Sigma^*}$.

Similarly, for KAT, there is a syntactic derivative

$$D_{\alpha p} : \mathsf{Exp} \to \mathsf{Exp} \qquad E_\alpha : \mathsf{Exp} \to 2$$

where now Exp is the set of KAT expressions over a set of primitive actions Σ and tests B. Again, these can be defined by induction on the structure of the expression.

For NetKAT, a similar syntactic coalgebra exists, which we now define. It is of type

$$D_{\alpha\beta} : \mathsf{Exp} \to \mathsf{Exp} \qquad E_{\alpha\beta} : \mathsf{Exp} \to 2,$$

where Exp is the set of reduced NetKAT expressions. It is defined inductively as follows:

$$D_{\alpha\beta}(p) = 0 \qquad D_{\alpha\beta}(b) = 0 \qquad D_{\alpha\beta}(\mathsf{dup}) = \alpha \cdot [\alpha = \beta]$$
$$D_{\alpha\beta}(e_1 + e_2) = D_{\alpha\beta}(e_1) + D_{\alpha\beta}(e_2)$$
$$D_{\alpha\beta}(e_1 e_2) = D_{\alpha\beta}(e_1) \cdot e_2 + \sum_\gamma E_{\alpha\gamma}(e_1) \cdot D_{\gamma\beta}(e_2)$$
$$D_{\alpha\beta}(e^*) = D_{\alpha\beta}(e) \cdot e^* + \sum_\gamma E_{\alpha\gamma}(e) \cdot D_{\gamma\beta}(e^*)$$

$$E_{\alpha\beta}(p) = [p = p_\beta] \qquad E_{\alpha\beta}(b) = [\alpha = \beta \le b]$$
$$E_{\alpha\beta}(\mathsf{dup}) = 0 \qquad E_{\alpha\beta}(e_1 + e_2) = E_{\alpha\beta}(e_1) + E_{\alpha\beta}(e_2)$$
$$E_{\alpha\beta}(e_1 e_2) = \sum_\gamma E_{\alpha\gamma}(e_1) \cdot E_{\gamma\beta}(e_2)$$
$$E_{\alpha\beta}(e^*) = [\alpha = \beta] + \sum_\gamma E_{\alpha\gamma}(e) \cdot E_{\gamma\beta}(e^*).$$

The definitions for * are circular, but we take the least fixpoint of the system of equations.

Matrix Representation. At this point, the reader will probably have noticed that many of the operations in the definitions of $D_{\alpha\beta}$ and $E_{\alpha\beta}$ resemble matrix operations. Indeed, regarding the types of the coalgebra operations as

$$\delta : X \to X^{\mathsf{At}\times\mathsf{At}} \qquad \varepsilon : X \to 2^{\mathsf{At}\times\mathsf{At}},$$

we can view $\delta(t)$ as an $\mathsf{At}\times\mathsf{At}$ matrix over X and $\varepsilon(t)$ as an $\mathsf{At}\times\mathsf{At}$ matrix over 2. If X happens to be a KAT, then the family of $\mathsf{At}\times\mathsf{At}$ matrices over X again forms a KAT, denoted $\mathsf{Mat}(\mathsf{At}, X)$, under the standard matrix operations [10]. Thus we have

$$\delta : X \to \mathsf{Mat}(\mathsf{At}, X) \qquad \varepsilon : X \to \mathsf{Mat}(\mathsf{At}, 2).$$

In this view, the syntactic coalgebra defined in §3.2 takes the following succinct form:

$$D(p) = 0 \qquad D(b) = 0 \qquad D(\mathsf{dup}) = J$$
$$D(e_1 + e_2) = D(e_1) + D(e_2)$$
$$D(e_1 e_2) = D(e_1) \cdot I(e_2) + E(e_1) \cdot D(e_2)$$
$$D(e^*) = E(e^*) \cdot D(e) \cdot I(e^*),$$

where $I(e)$ is the diagonal matrix with e on the main diagonal and J is the diagonal matrix with α on the main diagonal in position $\alpha\alpha$; and

$$E(p) = C_{\alpha_p} \qquad E(b) = J \cdot I(b)$$
$$E(\mathsf{dup}) = 0 \qquad E(e_1 + e_2) = E(e_1) + E(e_2)$$
$$E(e_1 e_2) = E(e_1) \cdot E(e_2)$$
$$E(e^*) = E(e)^*,$$

where C_α is the matrix with 1's in column α and 0's elsewhere. Thus E becomes a KAT homomorphism $E : \mathsf{Exp} \to \mathsf{Mat}(\mathsf{At}, 2)$.

Siumilarly, we can regard the set-theoretic coalgebra of §3.2 as having type

$$\delta : 2^U \to \mathsf{Mat}(\mathsf{At}, 2^U) \qquad \varepsilon : 2^U \to \mathsf{Mat}(\mathsf{At}, 2).$$

In this form, ε becomes a KAT homomorphism:

Lemma 1.

(i) $\varepsilon(1) = I$
(ii) $\varepsilon(A \cup B) = \varepsilon(A) + \varepsilon(B)$
(iii) $\varepsilon(A \cdot B) = \varepsilon(A) \cdot \varepsilon(B)$
(iv) $\varepsilon(A^*) = \varepsilon(A)^*$

Proof. These properties follow straightforwardly from the definitions in §3.2. For example, for (iii) and (iv), we have

$$\begin{aligned}\varepsilon(AB)_{\alpha\beta} &= [\alpha p_\beta \in AB] \\ &= [\exists\gamma\ \alpha p_\gamma \in A \wedge \gamma p_\beta \in B] \\ &= \sum_\gamma [\alpha p_\gamma \in A] \cdot [\gamma p_\beta \in B] \\ &= \sum_\gamma \varepsilon(A)_{\alpha\gamma} \cdot \varepsilon(B)_{\gamma\beta} \\ &= (\varepsilon(A) \cdot \varepsilon(B))_{\alpha\beta}\end{aligned}$$

$$\varepsilon(A^*) = \varepsilon(\textstyle\bigcup_n A^n) = \sum_n \varepsilon(A)^n = \varepsilon(A)^*.$$

□

Lemma 2.

(i) $\delta(\bigcup_n A_n) = \sum_n \delta(A_n)$
(ii) $\delta(AB) = \delta(A) \cdot I(B) + \varepsilon(A) \cdot \delta(B)$
(iii) $\delta(A^*) = \varepsilon(A^*) \cdot \delta(A) \cdot I(A^*)$

where $I(A)$ is the diagonal matrix with the set A on the main diagonal and $\varnothing$ elsewhere, and the matrix sum in (i) is componentwise union.

Proof. We argue (ii) and (iii) explicitly; (i) is straightforward from linearity.

For (ii), by definition we have

$$\delta_{\alpha\beta}(AB) = \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in AB\}.$$

In order that $\alpha p_\beta \ \mathsf{dup}\ x \in AB$, the string must the product of two reduced strings, one from A and one from B. Depending on which of these contains the first occurrence of dup, one of the following must occur:

- there exists γ such that $\alpha p_\beta \ \mathsf{dup}\ x = \alpha p_\gamma \cdot \gamma p_\beta \ \mathsf{dup}\ x$ with $\alpha p_\gamma \in A$ and $\gamma p_\beta \ \mathsf{dup}\ x \in B$; or
- there exist γ, y, and z such that $\alpha p_\beta \ \mathsf{dup}\ x = \alpha p_\beta \ \mathsf{dup}\ y p_\gamma \cdot \gamma z$ with $\alpha p_\beta \ \mathsf{dup}\ y p_\gamma \in A$, $\gamma z \in B$, and $x = y p_\gamma \gamma z$.

In the first case, we have $\varepsilon_{\alpha\gamma}(A) = 1$ and $\beta x \in \delta_{\gamma\beta}(B)$, hence $\beta x \in \varepsilon_{\alpha\gamma}(A) \cdot \delta_{\gamma\beta}(B)$. In the second case, we have $\beta y p_\gamma \in \delta_{\alpha\beta}(A)$ and $\gamma z \in B$, hence $\beta x = \beta y_\gamma \gamma z \in \delta_{\alpha\beta}(A) \cdot B$. Thus

$$\delta_{\alpha\beta}(AB) = \delta_{\alpha\beta}(A) \cdot B \ \cup \ \bigcup_\gamma \varepsilon_{\alpha\gamma}(A) \cdot \delta_{\gamma\beta}(B).$$

Abstracting over indices, we obtain the matrix equation (ii).

For (iii), we have from (i) and (ii) that

$$\begin{aligned}\delta(A^*) &= \delta(1 + AA^*) = 0 + \delta(AA^*) \\ &= \delta(A) \cdot I(A^*) + \varepsilon(A) \cdot \delta(A^*).\end{aligned}$$

The derivative is the least fixpoint of this equation, which by an axiom of KAT is the right-hand side of (iii). □

The following lemma says that G is a coalgebra morphism from the syntactic coalgebra (Exp, D, E) to $(2^U, \delta, \varepsilon)$ (G is the unique homomorphism into the final coalgebra).

Lemma 3.

(i) $G(D(e)) = \delta(G(e))$
(ii) $E(e) = \varepsilon(G(e))$

where G is extended componentwise to matrices.

Proof. By induction on e.

(i) For primitive p, b and dup,

$$\begin{aligned}G(D_{\alpha\beta}(p)) &= G(0) = \varnothing \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\gamma p \mid \gamma \in \mathsf{At}\}\} \\ &= \delta_{\alpha\beta}(\{\gamma p \mid \gamma \in \mathsf{At}\}) = \delta_{\alpha\beta}(G(p)) \\ G(D_{\alpha\beta}(b)) &= G(0) = \varnothing \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\beta p_\beta \mid \beta \le b\}\} \\ &= \delta_{\alpha\beta}(\{\beta p_\beta \mid \beta \le b\}) = \delta_{\alpha\beta}(G(b)). \\ G(D_{\alpha\beta}(\mathsf{dup})) &= G(\alpha \cdot [\alpha = \beta]) \\ &= \{\beta p_\beta \mid \alpha = \beta\} \\ &= \{\beta x \mid \alpha p_\beta \ \mathsf{dup}\ x \in \{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}\} \\ &= \delta_{\alpha\beta}(\{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}) \\ &= \delta_{\alpha\beta}(G(\mathsf{dup}))\end{aligned}$$

The case $e_1 + e_2$ is straightforward, since G, δ, and D are linear. For products, using Lemma 2(ii),

$$\begin{aligned}G(D(e_1e_2)) &= G(D(e_1) \cdot I(e_2)) + G(E(e_1) \cdot D(e_2)) \\ &= G(D(e_1)) \cdot G(I(e_2)) + G(E(e_1)) \cdot G(D(e_2)) \\ &= \delta(G(e_1)) \cdot I(G(e_2)) + \varepsilon(G(e_1)) \cdot \delta(G(e_2)) \\ &= \delta(G(e_1) \cdot G(e_2)) \\ &= \delta(G(e_1e_2))\end{aligned}$$

For star, the system defining $D(e^*)$ is

$$D(e^*) = D(e) \cdot I(e^*) + E(e) \cdot D(e^*)$$

whose least solution is

$$D(e^*) = E(e^*) \cdot D(e) \cdot I(e^*).$$

Using Lemma 2(iii),

$$\begin{aligned}G(D(e^*)) &= G(E(e^*) \cdot D(e) \cdot I(e^*)) \\ &= G(E(e^*)) \cdot G(D(e)) \cdot G(I(e^*)) \\ &= \varepsilon(G(e)^*) \cdot \delta(G(e)) \cdot I(G(e)^*) \\ &= \delta(G(e^*)).\end{aligned}$$

(ii) For p, b and dup,

$$\begin{aligned}E_{\alpha\beta}(p) &= [p = p_\beta] \\ &= \varepsilon_{\alpha\beta}(\{\gamma p \mid \gamma \in \mathsf{At}\}) = \varepsilon(G(p)). \\ E_{\alpha\beta}(b) &= J \cdot I(b) \\ &= [\alpha = \beta \le b] \\ &= \varepsilon_{\alpha\beta}(\{\alpha p_\alpha \mid \alpha \le b\}) \\ &= \varepsilon_{\alpha\beta}(G(b)). \\ E_{\alpha\beta}(\mathsf{dup}) &= 0 \\ &= \varepsilon_{\alpha\beta}(\{\gamma p_\gamma \ \mathsf{dup}\ p_\gamma \mid \gamma \in \mathsf{At}\}) \\ &= E_{\alpha\beta}(G(\mathsf{dup}))\end{aligned}$$

The case $e_1 + e_2$ is straightforward, since G, ε, and E are linear. For products, using Lemma 1(iii),

$$\begin{aligned}E_{\alpha\beta}(e_1e_2) &= \sum_\gamma E_{\alpha\gamma}(e_1) \cdot E_{\gamma\beta}(e_2) \\ &= (E(e_1) \cdot E(e_2))_{\alpha\beta} \\ &= (\varepsilon(G(e_1)) \cdot \varepsilon(G(e_2)))_{\alpha\beta} \\ &= (\varepsilon(G(e_1) \cdot G(e_2)))_{\alpha\beta} \\ &= \varepsilon_{\alpha\beta}(G(e_1e_2))\end{aligned}$$

For star, using Lemma 1(iv),

$$E(e^*) = E(e)^* = \varepsilon(G(e))^* = \varepsilon(G(e^*)).$$

□

4. Kleene's Theorem for NetKAT

In this section we show that a subset of U is $G(e)$ for some NetKAT expression e iff it is the set of strings accepted by some finite NetKAT automaton. This is the analog of Kleene's theorem for NetKAT.

4.1 From Automata to Expressions

Let $M = (S, \delta, \varepsilon, s)$ be a finite NetKAT automaton. Consider a graph H with nodes $(S \times \mathsf{At}) \cup \{\mathsf{halt}\}$ and labeled edges

$$\begin{aligned}&(u, \alpha) \xrightarrow{p_\beta \mathsf{dup}} (v, \beta), && \text{if } \delta_{\alpha\beta}(u) = v \\ &(u, \alpha) \xrightarrow{p_\beta} \mathsf{halt}, && \text{if } \varepsilon_{\alpha\beta}(u) = 1.\end{aligned}$$

We claim that for $x \in (P \cdot \mathsf{dup})^* \cdot P$,

$$(t, \alpha) \xrightarrow{x} \mathsf{halt} \ \Leftrightarrow \ \mathsf{Accept}(t, \alpha x). \tag{4.1}$$

This can be proved by induction on the length of x. For the basis,

$$(t, \alpha) \xrightarrow{p_\beta} \mathsf{halt} \ \Leftrightarrow \ \varepsilon_{\alpha\beta}(t) = 1 \ \Leftrightarrow \ \mathsf{Accept}(t, \alpha p_\beta).$$

For the induction step,

$$\begin{aligned}(t, \alpha) \xrightarrow{p_\beta \mathsf{dup} x} \mathsf{halt} \ &\Leftrightarrow \ \exists u \ (t, \alpha) \xrightarrow{p_\beta \mathsf{dup}} (u, \beta) \xrightarrow{x} \mathsf{halt} \\ &\Leftrightarrow \ \exists u \ \delta_{\alpha\beta}(t) = u \wedge \mathsf{Accept}(u, \beta x) \\ &\Leftrightarrow \ \mathsf{Accept}(\delta_{\alpha\beta}(t), \beta x) \\ &\Leftrightarrow \ \mathsf{Accept}(t, \alpha p_\beta \ \mathsf{dup}\ x).\end{aligned}$$

The set of labels of paths in H from (t, α) to halt is a regular subset of $(P \cdot \mathsf{dup})^* \cdot P$ and is described by a regular expression $e(t, \alpha)$. These expressions can be computed by taking the star of H considered as a square matrix. By (4.1), the set of strings accepted by M is the regular subset of U described by $e = \sum_\alpha \alpha \cdot e(s, \alpha)$.

As shown in [1], if $R(e) \subseteq U$, where R is the canonical interpretation of regular expressions as regular sets of strings, then $R(e) = G(e)$. We have shown

Theorem 1. *Let M be a finite NetKAT automaton. The set of strings in U accepted by M is $G(e)$ for some NetKAT expression e.*

4.2 From Expressions to Automata

To go in the other direction, we construct a finite NetKAT automaton M_e from an expression e. The states are NetKAT expressions modulo associativity, commutativity, and idempotence (ACI). The continuation and observation maps are the syntactic derivative introduced in §3.2. The start state is e.

Lemma 4. *The set accepted by M_e is $G(e)$.*

Proof. By Lemma 3, G is a coalgebra homomorphism from the syntactic coalagebra (Exp, D, E) to the set-theoretic coalgebra $(2^U, \delta, \varepsilon)$. Proceeding by induction on the length of the string,

$$\begin{aligned}
\mathsf{Accept}(e, \alpha p_\beta) &\Leftrightarrow E_{\alpha\beta}(e) = 1 \\
&\Leftrightarrow G(E_{\alpha\beta}(e)) = 1 \\
&\Leftrightarrow \varepsilon_{\alpha\beta}(G(e)) = 1 \\
&\Leftrightarrow \alpha p_\beta \in G(e), \\
\mathsf{Accept}(e, \alpha p_\beta \ \mathsf{dup}\ x) &\Leftrightarrow \mathsf{Accept}(D_{\alpha\beta}(e), \beta x) \\
&\Leftrightarrow \beta x \in G(D_{\alpha\beta}(e)) \\
&\Leftrightarrow \beta x \in \delta_{\alpha\beta}(G(e)) \\
&\Leftrightarrow \alpha p_\beta \ \mathsf{dup}\ x \in G(e).
\end{aligned}$$

□

It remains to show that M_e is finite. We do this by showing that e has only finitely many derivatives up to ACI.

4.2.1 Spines

For $A \subseteq \mathsf{Exp}$ and $e \in \mathsf{Exp}$, write $A \cdot e$ for $\{de \mid d \in A\}$ and write $e \cdot A$ for $\{ed \mid d \in A\}$.

A *spine* of e is an expression in Exp constructed from e as defined below. As we will show, the derivatives of e can be constructed from spines of e. There is one spine of e for each occurrence of dup in e. The set of spines of e is denoted $Sp(e)$ and is defined inductively:

$$\begin{aligned}
Sp(e_1 + e_2) &= Sp(e_1) \cup Sp(e_2) \\
Sp(e_1 e_2) &= Sp(e_1) \cdot e_2 \cup Sp(e_2) \\
Sp(e^*) &= Sp(e) \cdot e^* \\
Sp(\mathsf{dup}) &= \{1\} \\
Sp(b) &= Sp(p) = \varnothing.
\end{aligned}$$

It is easily shown that every element of $Sp(e)$ is of the form $1e_1e_2 \cdots e_n$, where the e_i are subterms of e, and that there is one spine of e for every occurrence of dup in e.

The following result says that derivatives of e are sums of spines of e.

Lemma 5. *For any α, β, the derivative $D_{\alpha\beta}(e)$ is a sum of terms of the form βd, where $d \in Sp(e)$.*

Proof. This can be proved by induction on the structure of e. Abusing notation slightly by representing sums as sets of terms,[1] we argue the case of products and star explicitly.

[1] This is a convenient abuse which we can take with impunity since we are working modulo ACI. The representation of the Brzozowski derivative in this form is often called the *Antimirov derivative*.

For products, we have

$$\begin{aligned}
D_{\alpha\beta}(e_1e_2) &= D_{\alpha\beta}(e_1) \cdot e_2 \cup \textstyle\bigcup_\gamma E_{\alpha\gamma}(e_1) \cdot D_{\gamma\beta}(e_2) \\
&\subseteq \beta \cdot Sp(e_1) \cdot e_2 \cup \beta \cdot Sp(e_2) \\
&= \beta \cdot (Sp(e_1) \cdot e_2 \cup Sp(e_2)) \\
&= \beta \cdot Sp(e_1e_2).
\end{aligned}$$

The induction hypothesis was used in the second step.

For star, we have

$$\begin{aligned}
D_{\alpha\beta}(e^*) &= \textstyle\bigcup_\gamma E_{\alpha\gamma}(e^*) \cdot D_{\gamma\beta}(e) \cdot e^* \\
&\subseteq \beta \cdot Sp(e) \cdot e^* \\
&= \beta \cdot Sp(e^*).
\end{aligned}$$

□

The next lemma shows that spines of spines of e are spines of e. Thus repeated derivatives do not introduce any new terms.

Lemma 6. *If $d \in Sp(e)$, then $Sp(\beta d) \subseteq Sp(e)$.*

Proof. For sums,

$$\begin{aligned}
& d \in Sp(e_1 + e_2) = Sp(e_1) \cup Sp(e_2) \\
& \Rightarrow d \in Sp(e_1) \text{ or } d \in Sp(e_2) \\
& \Rightarrow Sp(\beta d) \subseteq Sp(e_1) \text{ or } Sp(\beta d) \subseteq Sp(e_2) \\
& \Rightarrow Sp(\beta d) \subseteq Sp(e_1) \cup Sp(e_2) = Sp(e_1 + e_2).
\end{aligned}$$

For products,

$$\begin{aligned}
& d \in Sp(e_1e_2) = Sp(e_1) \cdot e_2 \cup Sp(e_2) \\
& \Rightarrow d \in Sp(e_1) \cdot e_2 \text{ or } d \in Sp(e_2) \\
& \Rightarrow (d = ce_2 \text{ and } c \in Sp(e_1)) \text{ or } d \in Sp(e_2) \\
& \Rightarrow (d = ce_2 \text{ and } Sp(\beta c) \subseteq Sp(e_1)) \\
& \quad \text{or } Sp(\beta d) \subseteq Sp(e_2) \\
& \Rightarrow Sp(\beta d) = Sp(\beta c e_2) = Sp(\beta c) \cdot e_2 \cup Sp(e_2) \\
& \quad \subseteq Sp(e_1) \cdot e_2 \cup Sp(e_2) = Sp(e_1e_2) \\
& \quad \text{or } Sp(\beta d) \subseteq Sp(e_2) \subseteq Sp(e_1e_2) \\
& \Rightarrow Sp(\beta d) \subseteq Sp(e_1e_2).
\end{aligned}$$

For star,

$$\begin{aligned}
& d \in Sp(e^*) = Sp(e) \cdot e^* \\
& \Rightarrow d = ce^* \text{ and } c \in Sp(e) \\
& \Rightarrow Sp(\beta d) = Sp(\beta c e^*) = Sp(\beta c) e^* \cup Sp(e^*) \\
& \quad \subseteq Sp(e) \cdot e^* \cup Sp(e^*) = Sp(e^*).
\end{aligned}$$

For dup,

$$\begin{aligned}
& d \in Sp(\mathsf{dup}) = \{1\} \\
& \Rightarrow d = 1 \\
& \Rightarrow Sp(\beta d) = Sp(\beta) = \varnothing \subseteq Sp(\mathsf{dup}).
\end{aligned}$$

We cannot have $d \in Sp(b)$ or $d \in Sp(p)$, since these sets are empty. □

Using these two lemmas, we can show that repeated derivatives of e can all be represented as sums of terms of the form βd, where $d \in Sp(e)$. Thus the number of derivatives of e is at most $|\mathsf{At}| \cdot 2^\ell$, where ℓ is the number of occurrences of dup in e. These can all be represented efficiently as an atom and a subset of $Sp(e)$. We have shown

Theorem 2. *For every NetKAT expression e, there is a deterministic NetKAT automaton M_e with at most $|\mathsf{At}| \cdot 2^\ell$ states accepting the set $G(e)$.*

5. Implementation

We have built a prototype implementation of our algorithm for deciding NetKAT equivalence in OCaml. Given a pair of NetKAT terms, it first uses Brzozowski derivatives to construct a pair of automata, and then checks whether the automata are bisimilar using a standard coinductive algorithm. Our implementation consists of 2095 lines of OCaml code and includes a parser, pretty printer, automata representation, and a simple interactive top-level loop. It also incorporates important enhancements and optimizations that avoid several potential sources of combinatorial blowup.

Challenges. The algorithms described in preceding sections are all formulated in terms of elements of the NetKAT language model, which contains complete tests and assignments. As such it would not be practical to implement these algorithms literally, since real-world NetKAT programs have on the order of 2^{250} distinct complete tests and assignments—one for each possible packet header value. Clearly constructing and iterating through such a large collection of values would be completely impractical. Instead, our implementation is based on clever representations of terms and matrices that exploit symmetry and sparseness, and only represents values that are relevant to the final outcome. Thus, unlike the algorithm presented in the original paper on NetKAT [1], our implementation avoids having to ever iterate through the entire universe in the common case. Of course, the fundamental decision problem is still PSPACE-complete, so the worst case computational complexity is unavoidable. However, in many cases of practical interest, the inputs are simple enough that our prototype quickly returns an answer.

NetKAT Representation. The first optimization used in our implementation is to represent the abstract syntax trees for NetKAT sums using sets. Derivatives built using such a representation are often called *Antimirov derivatives*. One advantage of this representation is that it gives addition modulo ACI essentially for free since we can use smart constructors to ensure that equivalent terms are represented using identical sets in OCaml. In addition, the elements of the sets are all spines (see §4.2.1), so the number of possible subsets is bounded by 2^k, where k is the number of occurrences of dup in the original term.

To build this set representation in OCaml, we define a pair of mutually recursive modules for terms and sets of terms:

```
module rec Term : sig
  type term =
  | Test of field * value
  | Assg of field * value
  | Dup
  | Plus of TermSet.t
  | Times of term list
  | ...
end = Term
and TermSet : sig
  include Set.S
  (* extra functions *)
  val bind : t -> (elt -> t) -> t
  val return : elt -> t
  ...
end with type elt = Term.term = struct
  include Set.Make (struct
    type t = Term.term
    let compare = Pervasives.compare
  end)
  (* extra functions *)
  let bind ts f = ...
  let return x = ...
  ...
end
```

Note that we also represent products as lists, which gives multiplication modulo associativity for free.

Right and Left Spines. Our implementation relies heavily on the notion of *spines* from §4.2.1. Recall that there is one spine for every occurrence of dup in e. We call these *right spines* because they consist of terms that occur to the right of the occurrence of dup.

```
let spines (e : term) : TermSet.t =
  let rec sp (e : term) : TermSet.t =
    match e with
    | Dup -> TermSet.return One
    | Plus ts -> TermSet.bind ts sp
    | ... in
  TermSet.map Ast.simplify (sp e)
```

There is a symmetric notion of a *left spine*, which also plays an important role in our implementation:

$$\begin{aligned} Sp(e_1 + e_2) &= Sp(e_1) \cup Sp(e_2) \\ Sp(e_1 e_2) &= e_1 \cdot Sp(e_2) \cup Sp(e_1) \\ Sp(e^*) &= e^* \cdot Sp(e) \\ Sp(\mathsf{dup}) &= \{1\} \\ Sp(b) &= Sp(p) = \varnothing. \end{aligned}$$

If σ denotes an occurrence of dup in a term e, let ℓ_σ and r_σ denote the left spine and right spine of that occurrence respectively. It is easy to show that the derivatives of e can be defined in terms of left and right spines as follows,

$$D_{\alpha\beta}(e) = \bigcup\{\beta r_\sigma \mid \sigma \text{ an occurrence of } \mathsf{dup},\ E_{\alpha\beta}(\ell_\sigma) = 1\}$$

or even more succinctly:

$$D(e) = \sum_\sigma E(\ell_\sigma) \cdot J \cdot I(r_\sigma), \tag{5.1}$$

where J is the diagonal matrix with α on the main diagonal in postion $\alpha\alpha$.

Moreover, since all of the spines of spines of e are spines of e, we need only find the left spines of all right spines once and for all. Hence, after calculating one derivative, the subsequent derivative will also be of the form (5.1). This means that we can compute the spines and 0-1 matrices $E(\ell_\sigma)$ in advance, and the matrices can be computed inductively using a sparse matrix representation described next.

Sparse Matrix Representation. We describe now how to calculate the 0-1 matrix $E(e)$ efficiently using a compact representation as sets of indices. Recall that the columns of a matrix are indexed by complete assignments and the rows by complete tests. Let $x_1, \ldots, x_n$ be the list of all field names appearing in the program. Let U_i be the universe of all values v associated with x_i in the program, either by a test $x_i = v$ or $x_i \neq v$ or by an assignment $x_i \leftarrow v$. Augment each universe U_i with a special value ? that denotes an unknown value. The set of all possible indices is $U_1 \times \cdots \times U_n$ for both rows and columns.

We representat 0-1 matrices using sets of pairs of sequences

$$A_1, \ldots, A_n; k_1, \ldots, k_n \tag{5.2}$$

where $A_i \subseteq U_i$ and $k_i \in U_i$. The sequence (5.2) represents the set of complete tests $A_1 \times \cdots \times A_n$, each followed by a complete assignment. The complete assignment corresponding to the complete test $m_1, \ldots, m_n \in A_1 \times \cdots \times A_n$ contains $x_i \leftarrow k_i$ if $k_i \neq ?$ and $x_i \leftarrow m_i$ if $k_i = ?$. This representation admits efficient matrix addition and multiplication without having to enumerate over the domain of all possible indices. Moreover, the primitive matrices are all straightforward to construct: the test $x_i = m_i$ is represented by

$$U_1, \ldots, U_{i-1}, \{m_i\}, U_{i+1}, \ldots, U_n; ?, \ldots, ?,$$

the test $x_i \neq m_i$ is represented by

$$U_1, \ldots, U_{i-1}, U_i - \{m_i\}, U_{i+1}, \ldots, U_n; \texttt{?}, \ldots, \texttt{?},$$

and the assignment $x_i \leftarrow m_i$ is represented by

$$U_1, \ldots, U_n; \texttt{?}, \ldots, \texttt{?}, m_i, \texttt{?}, \ldots, \texttt{?}.$$

Derivative. After calculating the spines and $E(e)$ matrices, the matrix $D(e)$ representing the Brzozowski derivative can be computed as follows: the $E(e)$ matrices are represented sparsely using sets of indices $\alpha\beta$ for which the corresponding matrix entry is 1. For each spine pair ℓ_σ, r_σ and nonzero element $\alpha\beta$ of $E(e)$, we add βr_σ to $D_{\alpha\beta}(e)$, as specified by 5.1. This approach exploits the sparseness of E so that we only need to consider those indices $\alpha\beta$ that might contribute a non-zero element to $D(e)$.

To further increase the compactness of $D(e)$, the complete assignment βr_σ is added to $D_{\alpha\beta}(e)$ only when βr_σ is nonzero. Let α_β be the complete test corresponding to β such that $\beta\alpha_\beta = \beta$. Let α_{r_σ} be a complete test such that $\alpha_{r_\sigma} r_\sigma = r_\sigma$. For βr_σ to be nonzero, α_β must be equal to some α_{r_σ}. The set of all α_{r_σ} can easily be extracted from $E(r_\sigma)$, it is the $m_1, \ldots, m_n \in A_1 \times \cdots \times A_n$ of the corresponding 0-1 matrix of $E(r_\sigma)$. Hence, taking advantage of our 0-1 matrix representation, the derivative can be more compactly described as:

$$D(e) = \sum_\sigma E(l_\sigma) \cdot E(r_\sigma)' \cdot J \cdot I(r_\sigma),$$

where

$$E(r_\sigma) = A_1, \cdots, A_n; k_1, \cdots, k_n$$
$$E(r_\sigma)' = A_1, \cdots, A_n; \texttt{?}, \cdots, \texttt{?}.$$

Bisimulation. The final step in the decision procedure uses a standard algorithm to test the bisimilarity of the automata constructed lazily using derivatives. Given two NetKAT terms e_1 and e_2, we first compare the matrices $E(e_1)$ and $E(e_2)$ and return false if they are not identical. Otherwise, we calculate the derivatives of e_1 and e_2 using D, and recursively check each of the resulting pairs. The algorithm halts when we have tested every possible derivative reachable transitively from the initial terms. Checking derivatives modulo ACI guarantees that the algorithm terminates. This coinductive algorithm can be implemented in almost linear time in the combined size of the automata using the union-find data structure [14]. Although our current prototype does not implement the up-to techniques used by Bonchi and Pous [4], we hope to incorporate these enhancements in the near future and expect they will yield further performance improvements.

Examples. We now present a series of small examples that illustrate the key algorithms and data structures used in our prototype implementation. As a first example, consider the following pair of NetKAT terms:

```
x = 3 ; y = 2 ; x := 7 + x = 4 ; y = 2 ; x := 7
```

and

```
(x = 3 + x = 4) ; y = 2 ; x := 7
```

These terms are clearly equivalent, as can be shown using the NetKAT axiom KA-SEQ-DIST-L, which distributes the test over the sum. To verify this fact using our bisimulation-based algorithm, we would first calculate the 0-1 matrices for the left spine, obtaining:

```
0 0 0 0 1 0 0 0          0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0          0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0   and    0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
```

Note that these matrices are identical and that our implementation automatically shrinks the universe into a small set, only representing the constants that actually appear in the program. In fact, our implementation would not even represent this matrix (unless asked to print it)—the sparse representation maintains just enough information to keep track of the non-zero entries. Next, the algorithm would systematically take all derivatives of each term. In this case, because neither term contains a dup, there are no non-zero derivatives and the algorithm halts immediately, and returns true.

For another example, consider

```
x = 1 ; x := 2 + x = 2 ; x := 3
```

and

```
x = 3 ; x := 2 + x = 2 ; x := 3
```

These terms are clearly not equivalent since the first matches packets whose x field is either 1 or 2, while the second matches packets whose x field is 2 or 3. In this case, the 0-1 matrices for the left spine are as follows:

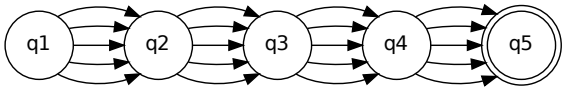
```
0 1 0 0          0 0 0 0
0 0 1 0   and    0 0 1 0
0 0 0 0          0 1 0 0
0 0 0 0          0 0 0 0
```

As these matrices are not identical, there exists a packet that distinguishes them so the algorithm immediately returns false.

As a final example, we illustrate the construction of a simple automaton on the following program,

```
x := 2 ; dup; x := 3; dup x:=4; dup ; x:= 5; dup
```

The NetKAT program consists of a sequence of assignments and dups. Accordingly, the automaton is a chain of states:



Note that there are transitions corresponding to each constant used in the program, and that only the final state is accepting.

Although these are not large examples, and we have not yet evaluated our prototype experimentally, based on our understanding of the algorithms and representations and our initial experiences using it on small examples we are optimistic that it will perform well.

6. Related Work

NetKAT [1] is the latest in a series of domain-specific languages for SDN programming developed as a part of the Frenetic project [12, 13, 25, 26]. NetKAT largely inherits its syntax, semantics, and application methodology from these earlier efforts but adds a complete deductive system and PSPACE decision procedure [1]. These new results in NetKAT build on the strong connection to earlier algebraic work in KA and KAT [20, 21, 23]. The present paper extends work on NetKAT further, developing the coalgebraic theory of the language and engineering an implementation of these ideas in an OCaml prototype. The overall result is the first practical implementation for deciding NetKAT equivalence.

The coalgebraic theories of KA and KAT and related systems have been studied extensively in recent years [9, 22, 30, 32], uncovering strong relationships between the algebraic/logical view of systems and the combinatorial/automata-theoretic view. We have exploited these ideas heavily in the development of NetKAT coalgebra and NetKAT automata. Finally, in our implementation, we

have borrowed many ideas and optimizations from the coalgebraic implementations of KA and KAT and other related systems [4, 5, 28] to provide enhanced performance, making automated decision feasible even in the face of PSPACE completeness.

A large number of languages for SDN programming have been proposed in recent years. Nettle [35] applies ideas from functional reactive programming to SDN programming, and focuses on making it easy to express dynamic programs using time-varying signals rather than event loops and callbacks as in most other systems. PANE [11] exposes an interface that allows individual hosts in a network to request explicit functionality such as increased bandwidth for a large bulk transfer or bounded latency for a phone call. Internally PANE uses hierarchical tables to represent and manage the set of requests and a compiler inspired by NetCore [25]. Maple [36] provides a high-level programming interface that enables programmers to express network programs directly in Java, using a special library to match and modify packet headers. Under the hood, the Maple compiler builds up representations of network traffic flows using a tree structure and then compiles these to hardware-level forwarding rules. Several different network programming languages based on logic programming have been proposed including NDLog [24] and FlowLog [27]. The key difference between all of these languages and the system presented in this paper is that NetKAT has a sound and complete deductive theory and supports automated reasoning about program equivalence.

Lastly, there is a growing body of work focused on applications of formal methods ranging from lightweight testing to full-blown verification to SDN. The NICE [8] tool uses a model checker and symbolic execution to find bugs in network programs written in Python. Automatic Test Packet Generation [38] constructs a set of packets that provide coverage for a given network-wide configuration. Retrospective Causal Inference [31] uses techniques based on delta debugging to reduce bugs to minimal input sequences. The VeriCon [2] system uses first-order logic and a notion of admissible topologies to automatically check network-wide properties. It uses the Z3 SMT solver as a back-end decision procedure. Several different systems have proposed techniques for checking network reachability properties including Xie et al. [37], Header Space Analysis [16], and VeriFlow[17]. These tools either translate reachability problems into problem instances for other tools, or they use custom decision procedures that extend basic satisfiability checking or ternary simulation with domain-specific optimizations to obtain improved performance. Compared to these tools, NetKAT is unique in its focus on algebraic and coalgebraic structure of network programs. Moreover, as shown in the original NetKAT paper, many properties of interest including reachability can be reduced to equivalence.

7. Conclusion

This paper develops the coalgebraic theory of NetKAT and leverages this theory to design a new decision procedure based on checking bisimulation between finite automata. The coalgebraic theory includes a definition of NetKAT automata, a variant of the Brzozowski derivative for NetKAT, and a version of Kleene's theorem relating NetKAT expressions and NetKAT automata. A novel aspect of the theory is the concise representation of the Brzozowski derivative in terms of matrices, which appears to be new and particular to NetKAT. We have given a new algorithm for deciding equivalence of NetKAT expressions and a full implementation which improves on the naive algorithm of [1]. The new algorithm holds much promise for feasibility based on reported experience with related systems [4, 5, 28].

Initial experiments with our implementation are very promising, however the experimental evaluation is still in the preliminary stages. For the future, we intend to continue to make further enhancements and perform extensive testing on practical examples from [1]. A straightforward extension is to incorporate in our algorithm well-studied enhancements to the bisimulation construction [4, 29] which will play a key role in the scalability of the equivalence checker. We also plan to explore extending alternative algorithms for deciding equivalence of KAT expressions [6, 33]. Another possible research direction is to study nondeterministic NetKAT automata, which will provide more compact representations of behaviors, and algorithms to decide language equivalence such as Brzozowski's algorithm [3] or extensions of Hopcroft and Karp's algorithm [4]. A long-term goal is to eventually deploy our algorithm in an SDN network management tool such as Merlin [34].

Acknowledgments. The authors wish to thank Arjun Guha, Andrew Myers, Mark Reitblatt, Ross Tate, Konstantinos Mamouras, and the rest of the Cornell PLDG for many insightful discussions and helpful comments.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'14)*, pages 113–126, San Diego, California, USA, January 2014. ACM.

[2] Thomas Ball, Nikolaj Bjorner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, , Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, 2014. To appear.

[3] Filippo Bonchi, Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. Brzozowski's algorithm (co)algebraically. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics*, volume 7230 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 2012.

[4] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. 40th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL '13, pages 457–468. ACM, 2013.

[5] Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1:16):1–42, 2012.

[6] Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. On the average size of glushkov and equation automata for kat expressions. In Leszek Gasieniec and Frank Wolter, editors, *FCT*, volume 8070 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2013.

[7] Janusz A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comput. Mach.*, 11:481–494, 1964.

[8] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[9] Hubie Chen and Riccardo Pucella. A coalgebraic approach to Kleene algebra with tests. *Electronic Notes in Theoretical Computer Science*, 82(1), 2003.

[10] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, Computer Science Department, Cornell University, July 1996.

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

[12] 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.

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

[14] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[15] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

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

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

[18] Teemu Koponen, Keith Amidon, Peter Balland, Martn Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[19] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

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

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

[22] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[23] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

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

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

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

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

[28] Damien Pous. Relational algebra and kat in coq, February 2013. Available at `http://perso.ens-lyon.fr/damien.pous/ra`.

[29] Jurriaan Rot, Marcello M. Bonsangue, and Jan J. M. M. Rutten. Coalgebraic bisimulation-up-to. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy R. Nawrocki, and Harald Sack, editors, *SOFSEM*, volume 7741 of *Lecture Notes in Computer Science*, pages 369–381. Springer, 2013.

[30] Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

[31] Robert Colin Scott, Andreas Wundsam, Kyriakos Zarifis, and Scott Shenker. What, Where, and When: Software Fault Localization for SDN. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[32] Alexandra Silva. *Kleene Coalgebra*. PhD thesis, University of Nijmegen, 2010.

[33] Alexandra Silva. Position automata for kleene algebra with tests. *Sci. Ann. Comp. Sci.*, 22(2):367–394, 2012.

[34] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the Network with Merlin. In *HotNets*, November 2013. To appear.

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

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

[37] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[38] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, 2012.