Kleene Algebra Modulo Theories

RYAN BECKETT, Princeton University
ERIC CAMPBELL, Pomona College
MICHAEL GREENBERG, Pomona College

Kleene algebras with tests (KATs) offer sound, complete, and decidable equational reasoning about regularly structured programs. Interest in KATs has increased greatly since NetKAT demonstrated how well extensions of KATs with domain-specific primitives and extra axioms apply to computer networks. Unfortunately, extending a KAT to a new domain by adding custom primitives, proving its equational theory sound and complete, and coming up with efficient automata-theoretic implementations is still an expert's task.

We present a general framework for deriving KATs we call *Kleene algebra modulo theories*: given primitives and a notion of state, we can automatically derive a corresponding KAT's semantics, prove its equational theory sound and complete with respect to a tracing semantics, use term normalization from the completeness proof to create a decision procedure for equivalence checking, and formalize an automata-based equivalence checking procedure as well. Our framework is based on *pushback*, a generalization of weakest preconditions that specifies how predicates and actions interact. We offer several case studies, showing plain theories (natural numbers, bitvectors, NetKAT) along with compositional theories (products, temporal logic, and sets). We are able to derive several results from the literature. Finally, we provide an OCaml implementation of both decision procedures that closely matches the theory: with only a few declarations, users can automatically compose KATs with complete decision procedures. We offer a fast path to a "minimum viable model" for those wishing to explore KATs formally or in code.

1 INTRODUCTION

Kleene algebras with tests (KATs) provide a powerful framework for reasoning about regularly structured programs. Modeling simple programs with while loops, KATs can handle a variety of analysis tasks [2, 7, 12–14, 36] and typically enjoy sound, complete, and decidable equational theories. Interest in KATs has increased recently as they have been applied to the domain of computer networks: NetKAT, a language for programming and verifying Software Defined Networks (SDNs), was the first remarkably successful extension of KAT [1], followed by many other variations and extensions [4, 8, 23, 37, 38, 48].

Considering KAT's success in networks, we believe other domains would benefit from programming languages where program equivalence is decidable. However, extending a KAT for a particular domain remains a challenging task even for experts familiar with KATs and their metatheory. To build a custom KAT, experts must craft custom domain primitives, derive a collection of new domain-specific axioms, prove the soundness and completeness of the resulting algebra, and implement a decision procedure. For example, NetKAT's theory and implementation was developed over several papers [1, 24, 51], after a long series of papers that resembled, but did not use, the KAT framework [22, 30, 39, 44]. Yet another challenge is that much of the work on KATs applies only to abstract, purely propositional KATs, where the actions and predicates are not governed by a set of domain-specific equations but are left abstract [16, 34, 40, 43]. Propositional KATs have limited applicability for domain-specific reasoning because domain-specific knowledge must be encoded manually as additional equational assumptions. In the presence of such equational assumptions, program equivalence becomes undecidable in general [12]. As a result, decision procedures have limited support for reasoning over domain-specific primitives and axioms [12, 32].

We believe domain-specific KATs will find more general application when it becomes possible to cheaply build and experiment with them. Our goal in this paper is to democratize KATs, offering

Authors' addresses: Ryan Beckett, Princeton University, rbeckett@cs.princeton.edu; Eric Campbell, Pomona College, ehc02013@mymail.pomona.edu; Michael Greenberg, Pomona College, michael@cs.pomona.edu.

a general framework for automatically deriving sound, complete, and decidable KATs for client theories. The proof obligations of our approach are relatively mild and our approach is *compositional*: a client can compose smaller theories to form larger, more interesting KATs than might be tractable by hand. In addition to the equivalence decision procedure that comes from our completeness proof's normalization routine, our theoretical framework has an automata theory that we prove correct. Our OCaml implementation allows users to compose a KAT with both decision procedures from small theory specifications. The automata are not only for verification, of course, they are useful for a variety of tasks such as compiling KATs to different implementations [8, 51]. We offer a fast path to a "minimum viable model" for those wishing to explore KATs formally or in code.

1.1 What is a KAT?

From a bird's-eye view, a Kleene algebra with tests is a first-order language with loops (the Kleene algebra) and interesting decision making (the tests). Formally, a KAT consists of two parts: a Kleene algebra $\langle 0, 1, +, \cdot, ^* \rangle$ of "actions" with an embedded Boolean algebra $\langle 0, 1, +, \cdot, \neg \rangle$ of "predicates". KATs capture While programs: the 1 is interpreted as skip, $\cdot$ as sequence, + as branching, and * for iteration. Simply adding opaque actions and predicates gives us a While-like language, where our domain is simply traces of the actions taken. For example, if α and β are predicates and π and ρ are actions, then the KAT term $\alpha \cdot \pi + \neg\alpha \cdot (\beta \cdot \rho)^* \cdot \neg\beta \cdot \pi$ defines a program denoting two kinds of traces: either α holds and we simply run π, or α doesn't hold, and we run ρ until β no longer holds and then run π. i.e., the set of traces of the form $\{\pi, \rho^*\pi\}$. Translating the KAT term into a While program, we write: `if` α `then` π `else { while` β `do {` ρ `};` π `}`. Moving from a While program to a KAT, consider the following program—a simple loop over two natural-valued variables `i` and `j`:

```
assume i < 50
while (i < 100) { i := i + 1; j := j + 2 }
assert j > 100
```

To model such a program in KAT, one replaces each concrete test or action with an abstract representation. Let the atomic test α represent the test $\mathtt{i} < 50$, β represent $\mathtt{i} < 100$, and γ represent $\mathtt{j} > 100$; the atomic actions p and q represent the assignments $\mathtt{i} := \mathtt{i} + 1$ and $\mathtt{j} := \mathtt{j} + 2$, respectively. We can now write the program as the KAT expression $\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg\beta \cdot \gamma$. The complete equational theory of KAT makes it possible to reason about program transformations and decide equivalence between KAT terms. For example, KAT's theory can prove that the assertion $j > 100$ must hold after running the while loop by proving that the set of traces where this does not hold is empty:

$$\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg\beta \cdot \neg\gamma \;\equiv\; 0$$

or that the original loop is equivalent to its unfolding:

$$\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg\beta \cdot \gamma \;\equiv\; \alpha \cdot (1 + \beta \cdot p \cdot q) \cdot (\beta \cdot p \cdot q \cdot \beta \cdot p \cdot q)^* \cdot \neg\beta \cdot \gamma$$

Unfortunately, KATs are naïvely propositional: the algebra understands nothing of the underlying domain or the semantics of the abstract predicates and actions. For example, the fact that $(\mathtt{j} := \mathtt{j} + 2 \cdot \mathtt{j} > 200) \equiv (\mathtt{j} > 198 \cdot \mathtt{j} := \mathtt{j} + 2)$ does not follow from the KAT axioms and must be added manually to any proof as an equational assumption. Yet the ability to reason about the equivalence of programs in the presence of particular domains is important for many real programs and domain-specific languages. To allow for reasoning with respect to a particular domain (e.g., the domain of natural numbers with addition and comparison), one typically must extend KAT with additional axioms that capture the domain-specific behavior [1, 4, 8, 29, 35].

Unfortunately, it remains an expert's task to extend the KAT with new domain-specific axioms, provide new proofs of soundness and completeness, and develop the corresponding implementation.

As an example of such a domain-specific KAT, NetKAT showed how packet forwarding in computer networks can be modeled as simple While programs. Devices in a network must drop or permit packets (tests), update packets by modifying their fields (actions), and iteratively pass packets to and from other devices (loops). NetKAT extends KAT with two actions and one predicate: an action to write to packet fields, $f \leftarrow v$, where we write value v to field f of the current packet; an action dup, which records a packet in a history log; and a field matching predicate, $f = v$, which determines whether the field f of the current packet is set to the value v. Each NetKAT program is denoted as a function from a packet history to a set of packet histories. For example, the program:

$$\mathsf{dstIP} \leftarrow 192.168.0.1 \cdot \mathsf{dstPort} \leftarrow 4747 \cdot \mathsf{dup}$$

takes a packet history as input, updates the current packet to have a new destination IP address and port, and then records the current packet state. The original NetKAT paper defines a denotational semantics not just for its primitive parts, but for the various KAT operators; they explicitly restate the KAT equational theory along with custom axioms for the new primitive forms, prove the theory's soundness, and then devise a novel normalization routine to reduce NetKAT to an existing KAT with a known completeness result. Later papers [24, 51] then developed the NetKAT automata theory used to compile NetKAT programs into forwarding tables and to verify networks. NetKAT's power comes at a cost: one must prove metatheorems and develop an implementation—a high barrier to entry for those hoping to apply KAT in their domain.

We aim to make it easier to define new KATs. Our theoretical framework and its corresponding implementation allow for quick and easy composition of sound and complete KATs with normalization-based and automata-theoretic decision procedures when given arbitrary domain-specific theories. Our framework, which we call Kleene algebras modulo theories (KMTs), allows us to derive metatheory and implementation for KATs based on a given theory. KMTs obviate the need to deeply understand KAT metatheory and implementation for a large class of extensions; a variety of higher-order theories allow language designers to compose new KATs from existing ones, allowing them to rapidly prototype their KAT theories.

1.2 Using our framework: obligations for client theories

Our framework takes a *client theory* and produces a KAT, but what must one provide in order to know that the generated KAT is deductively complete, or to derive an implementation? We require, at a minimum, a description of the theory's predicates and actions along with how these apply to some notion of state. We call these parts the *client theory*; the client theory's predicates and actions are *primitive*, as opposed to those built with the KAT's composition operators. We call the resulting KAT a *Kleene algebra modulo theory* (KMT). Deriving a trace-based semantics for the KMT and proving it sound isn't particularly hard—it amounts to "turning the crank". Proving the KMT is complete and decidable, however, can be much harder. We take care of much of the difficulty, lifting simple operations in the client theory generically to KAT.

Our framework hinges on an operation relating predicates and operations called *pushback*, first used to prove relative completeness for Temporal NetKAT [8]. Pushback is a generalization of weakest preconditions. Given a primitive action π and a primitive predicate α, the client theory must be able to compute weakest preconditions, telling us how to go from $\pi \cdot \alpha$ to some set of terms: $\sum_{i=0}^{n} \alpha_i \cdot \pi = \alpha_0 \cdot \pi + \alpha_1 \cdot \pi + \dots$. That is, the client theory must be able to take any of its primitive tests and "push it back" through any of its primitive actions. Given the client's notion of weakest preconditions, we can alter programs to take an *arbitrary* term and normalize it into a form where *all* of the predicates appear only at the front of the term, a convenient representation both for our completeness proof (Sec. 2.4) and our automata-theoretic implementation (Secs. 4 and 5).

The client theory's pushback should have two properties: it should be sound, (i.e., the resulting expression is equivalent to the original one); and none of the resulting predicates should be any bigger than the original predicates, by some measure (see Sec. 2). If the pushback has these two properties, we can use it to define a normal form for the KMT generated from the client theory—and we can use that normal form to prove that the resulting KMT is complete and decidable.

As an example, in NetKAT, for different fields f and f', we can use the network axioms to derive the equivalence: $(f \leftarrow v \cdot f' = v') \equiv (f' = v' \cdot f \leftarrow v)$, which satisfies the pushback requirements. For Temporal NetKAT, which adds rich temporal predicates such as $\Diamond \bigcirc (\mathsf{dstPort} = 4747)$ (the destination port was 4747 at some point before the previous state), we can use the domain axioms to prove the equivalence $(f \leftarrow v \cdot \Diamond \bigcirc a) \equiv (\Diamond \bigcirc a + a) \cdot f \leftarrow v$, which also satisfies the pushback requirements of equivalence and non-increasing measure.

Formally, the client must provide the following for our normalization routine (part of completeness): primitive tests and actions (α and π), semantics for those primitives (states σ and functions pred and act), a function identifying each primitive's subterms (sub), a weakest precondition relation (WP) justified by sound domain axioms ($\equiv$), and restrictions on WP term size growth.

The client's domain axioms extend the standard KAT equations to explain how the new primitives behave. In addition to these definitions, our client theory incurs a few proof obligations: $\equiv$ must be sound with respect to the semantics; the pushback relation should never push back a term that's larger than the input; the pushback relation should be sound with respect to $\equiv$; we need a satisfiability checking procedure for a Boolean algebra extended with the primitive predicates. Given these things, we can construct a sound and complete KAT with an automata-theoretic implementation.

1.3 Example: incrementing naturals

We can model programs like the While program over `i` and `j` from earlier by introducing a new client theory for natural numbers (Fig. 1). First, we extend the KAT syntax with actions $x := n$ and inc_x (increment x) and a new test $x > n$ for variables x and natural number constants n. First, we define the client semantics. We fix a set of variables, $\mathcal{V}$, which range over natural numbers, and the program state σ maps from variables to natural numbers. Primitive actions and predicates are interpreted over the state σ by the act and pred functions (where t is a trace of states).

Proof obligations. The WP relation provides a way to compute the weakest precondition for any primitive action and test. For example, the weakest precondition of $\mathsf{inc}_x \cdot x > n$ is $x > n - 1$ when n is not zero. We must have domain axioms to justify the weakest precondition relation. For example, the domain axiom: $\mathsf{inc}_x \cdot (x > n) \equiv (x > n - 1) \cdot \mathsf{inc}_x$ ensures that weakest preconditions for inc_x are modeled by the equational theory. The other axioms are used to justify the remaining weakest preconditions that relate other actions and predicates. Additional axioms that do not involve actions, such as $\neg(x > n) \cdot (x > m) \equiv 0$, are included to ensure that the predicate fragment of IncNat is complete in isolation. The deductive completeness of the model shown here can be reduced to Presburger arithmetic.

For the relative ease of defining IncNat, we get real power—we've extended KAT with unbounded state! It is sound to add other operations to IncNat, like multiplication or addition by a scalar. So long as the operations are monotonically increasing and invertible, we can still define a WP and corresponding axioms. It is *not* possible, however, to compare two variables directly with tests like $x = y$—to do so would not satisfy the requirement that weakest precondition does not grow the size of a test. It would be bad if it did: the test $x = y$ can encode context-free languages! The

Syntax

$$\begin{array}{lcl} \alpha & ::= & x > n \\ \pi & ::= & \mathsf{inc}_x \mid x := n \\ \mathsf{sub}(x > n) & = & \{x > m \mid m \leq n\} \end{array}$$

Semantics

$$\begin{array}{lcl} n \in \mathbb{N} & & x \in \mathcal{V} \\ \mathsf{State} & = & \mathcal{V} \rightarrow \mathbb{N} \\ \mathsf{pred}(x > n, t) & = & \mathsf{last}(t)(x) > n \\ \mathsf{act}(\mathsf{inc}_x, \sigma) & = & \sigma[x \mapsto \sigma(x) + 1] \\ \mathsf{act}(x := n, \sigma) & = & \sigma[x \mapsto n] \end{array}$$

Weakest precondition

$$\begin{array}{l} x := n \cdot (x > m) \text{ WP } (n > m) \\ \mathsf{inc}_y \cdot (x > n) \text{ WP } (x > n) \\ \mathsf{inc}_x \cdot (x > n) \text{ WP } (x > n - 1) \\ \qquad\qquad \text{when } n \neq 0 \\ \mathsf{inc}_x \cdot (x > 0) \text{ WP } 1 \end{array}$$

Axioms

$$\begin{array}{ll} \neg(x > n) \cdot (x > m) \equiv 0 \text{ when } n \leq m & \textsc{GT-Contra} \\ x := n \cdot (x > m) \equiv (n > m) \cdot x := n & \textsc{Asgn-GT} \\ (x > m) \cdot (x > n) \equiv (x > \max(m, n)) & \textsc{GT-Min} \\ \mathsf{inc}_y \cdot (x > n) \equiv (x > n) \cdot \mathsf{inc}_y & \textsc{GT-Comm} \\ \mathsf{inc}_x \cdot (x > n) \equiv (x > n - 1) \cdot \mathsf{inc}_x \text{ when } n > 0 & \textsc{Inc-GT} \\ \mathsf{inc}_x \cdot (x > 0) \equiv \mathsf{inc}_x & \textsc{Inc-GT-Z} \end{array}$$

Fig. 1. IncNat, increasing naturals

(inadmissible!) term $x := 0 \cdot y := 0; (\mathsf{inc}_x)^* \cdot (\mathsf{inc}_y)^* \cdot x = y$ describes programs with balanced increments of x and y. For the same reason, we cannot safely add a decrement operation dec_x. Either of these would allow us to define counter machines, leading inevitably to undecidability.

Implementation. Users implement KMT's client theories by defining OCaml modules; users give the types of actions and tests along with functions for parsing, computing subterms, calculating weakest preconditions for primitives, mapping predicates to an SMT solver, and deciding predicate satisfiability (see Sec. 5 for more detail).

Our example implementation starts by defining a new, recursive module called `IncNat`. Recursive modules allow the author of the module to access the final KAT functions and types derived after instantiating KA with their theory within their theory's implementation. For example, the module `K` on the second line gives us a recursive reference to the resulting KMT instantiated with the `IncNat` theory; such self-reference is key for higher-order theories, which must embed KAT predicates inside of other kinds of predicates (Sec. 3). The user must define two types: *a* for tests and *p* for actions. Tests are of the form $x > n$ where variable names are represented with `string`s, and values with OCaml `int`s. Actions hold either the variable being incremented (inc_x) or the variable and value being assigned ($x := n$).

```
type a = Gt of string * int     (* alpha ::= x > n *)
type p = Increment of string   (* pi    ::= inc x *)

module rec IncNat : THEORY with type A.t = a and type P.t = p = struct
 (* generated KMT, for recursive use *)
 module K = KAT (IncNat)
 (* boilerplate necessary for recursive modules, hashconsing *)
 module P : CollectionType with type t = p = struct ... end
 module A : CollectionType with type t = a = struct ... end
 (* extensible parser; pushback; subterms of predicates *)
 let parse name es = ...
 let push_back p a =
  match (p,a) with
  | (Increment x, Gt (_, j)) when 1 > j → PSet.singleton ~cmp:K.Test.compare (K.one ())
```

```
    | (Increment x, Gt (y, j)) when x = y →
      PSet.singleton ~cmp:K.Test.compare (K.theory (Gt (y, j − 1)))
    | (Assign (x,i), Gt (y,j)) when x = y →
      PSet.singleton ~cmp:K.Test.compare (if i > j then K.one () else K.zero ())
    | _ → PSet.singleton ~cmp:K.Test.compare (K.theory a)
  let rec subterms x =
   match x with
    | Gt (_, 0) → PSet.singleton ~cmp:K.Test.compare (K.theory x)
    | Gt (v, i) → PSet.add (K.theory x) (subterms (Gt (v, i − 1)))
  (* decision procedure for the predicate theory *)
  let satisfiable (a: K.Test.t) = ...
end
```

The first function, parse, allows the library author to extend the KAT parser (if desired) to include new kinds of tests and actions in terms of infix and named operators. The other functions, subterms and push_back, follow from the KMT theory directly. Finally, the user must implement a function that decides satisfiability of theory tests.

The implementation obligations—syntactic extensions, subterms functions, WP on primitives, a satisfiability checker for the test fragment—mirror our formal development. We offer more client theories in Sec. 3 and more detail on the implementation in Sec. 5.

1.4 Contributions

We claim the following contributions:

- A compositional framework for defining KATs and proving their metatheory, with a novel development of the normalization procedure used in completeness (Sec. 2) and a new KAT theorem (Pushback-Neg). Completeness yields a decision procedure based on normalization.
- Case studies of this framework (Sec. 3), several of which reproduce results from the literature, and several of which are new: base theories (e.g., naturals, bitvectors [29], networks), and more importantly, compositional, higher-order theories (e.g., sets and LTL_f). As an example, we define Temporal NetKAT compositionally [8] by applying the theory of LTL_f to a theory of NetKAT; doing so strengthens Temporal NetKAT's completeness result.
- An automata-theoretic account of our proof technique, proven correct and applicable to compilation and equivalence checking for, e.g., NetKAT (Sec. 4).
- An implementation of KMTs (Sec. 5) mirroring our proofs; we derive two equivalence decision procedures for client theories from just a few definitions, one based on our normalization routine and one using automata. Our implementation is efficient enough for experimentation with small programs (Sec. 6).

Finally, our framework offers a new way in for those looking to work with KATs. Researchers comfortable with inductive relations from, e.g., type theory and semantics, will find a familiar friend in pushback, our generalization of weakest preconditions—we define it as an inductive relation. To restate our contributions for readers more deeply familiar with KAT: Our framework is similar to Schematic KAT, a KAT extended with first order theories. However, Schematic KAT is incomplete in general. Our framework shows that a subset of Schematic KATs is complete—those with tracing semantics and a monotonic pushback.

Predicates $\mathcal{T}^*_{\text{pred}}$

a, b	$::=$	0	*additive identity*
	$\mid$	1	*multiplicative identity*
	$\mid$	$\neg a$	*negation*
	$\mid$	$a + b$	*disjunction*
	$\mid$	$a \cdot b$	*conjunction*
	$\mid$	α	*primitive predicates* ($\mathcal{T}_\alpha$)

Actions

p, q	$::=$	a	*embedded predicates*
	$\mid$	$p + q$	*parallel composition*
	$\mid$	$p \cdot q$	*sequential composition*
	$\mid$	p^*	*Kleene star*
	$\mid$	π	*primitive actions* ($\mathcal{T}_\pi$)

Fig. 2. $\mathcal{T}^*$: generalized KAT syntax over a client theory $\mathcal{T}$ (client parts highlighted)

2 THE KMT FRAMEWORK

The rest of our paper describes how our framework takes a client theory and generates a KAT. We emphasize that you need not understand the following mathematics to use our framework—we do it once and for all, so you don't have to. While we have striven to make this section accessible to non-expert readers, those completely new to KATs may do well to skip our discussion of pushback (Sec. 2.3.2 on) and read our case studies (Sec. 3). We highlight anything the client theory must provide.

We derive a KAT $\mathcal{T}^*$ (Fig. 2) on top of a client theory $\mathcal{T}$ where $\mathcal{T}$ has two fundamental parts—predicates $\alpha \in \mathcal{T}_\alpha$ and actions $\pi \in \mathcal{T}_\pi$. These are the *primitives* of the client theory. We refer to the Boolean algebra over the client theory as $\mathcal{T}^*_{\text{pred}} \subseteq \mathcal{T}^*$.

Our framework can provide results for $\mathcal{T}^*$ in a pay-as-you-go fashion: given a notion of state and an interpretation for the predicates and actions of $\mathcal{T}$, we derive a trace semantics for $\mathcal{T}^*$ (Sec. 2.1); if $\mathcal{T}$ has a sound equational theory with respect to our semantics, so does $\mathcal{T}^*$ (Sec. 2.2); if $\mathcal{T}$ has a complete equational theory with respect to our semantics, and satisfies certain weakest precondition requirements, then $\mathcal{T}^*$ has a complete equational theory (Sec. 2.4); and finally, with just a few lines of code defining the structure of $\mathcal{T}$, we can provide two decision procedures for equivalence (Sec. 5): one using the normalization routine from completeness (Sec. 2.4) and one using automata (Sec. 4).

The key to our general, parameterized proof is a novel *pushback* operation that generalizes weakest preconditions (Sec. 2.3.2): given an understanding of how to push primitive predicates back to the front of a term, we can normalize terms for our completeness proof (Sec. 2.4).

2.1 Semantics

The first step in turning the client theory $\mathcal{T}$ into a KAT is to define a semantics (Fig. 3). We can give any KAT a *trace semantics*: the meaning of a term is a trace t, which is a non-empty list of log entries l. Each *log entry* records a state σ and (in all but the initial state) a primitive action π. The client assigns meaning to predicates and actions by defining a set of states State and two functions: one to determine whether a predicate holds (pred) and another to determine an action's effects (act). To run a $\mathcal{T}^*$ term on a state σ, we start with an initial state $\langle \sigma, \bot \rangle$; when we're done, we'll have a set of traces of the form $\langle \sigma_0, \bot \rangle \langle \sigma_1, \pi_1 \rangle \ldots$, where $\sigma_i = \text{act}(\pi_i, \sigma_{i-1})$ for $i > 0$. (A similar semantics shows up in Kozen's application of KAT to static analysis [32].)

The client's pred function takes a primitive predicate α and a trace — predicates can examine the entire trace — returning true or false. When the pred function returns true, we return the singleton set holding our input trace; when pred returns false, we return the empty set. (Composite predicates follow this same pattern, always returning either a singleton set holding their input trace or the empty set.) It's acceptable for the pred function to recursively call the denotational

Trace definitions

$$\begin{array}{rclcl} \sigma & \in & \mathsf{State} & & \\ l & \in & \mathsf{Log} & ::= & \langle \sigma, \bot \rangle \mid \langle \sigma, \pi \rangle \\ t & \in & \mathsf{Trace} & = & \mathsf{Log}^+ \end{array}$$

$$\begin{array}{rcl} \mathsf{pred} & : & \mathcal{T}_\alpha \times \mathsf{Trace} \to \{\mathsf{true}, \mathsf{false}\} \\ \mathsf{act} & : & \mathcal{T}_\pi \times \mathsf{State} \to \mathsf{State} \end{array}$$

Trace semantics $\boxed{[\![-]\!] : \mathcal{T}^* \to \mathsf{Trace} \to \mathcal{P}(\mathsf{Trace})}$

$$\begin{array}{rcl} [\![0]\!](t) & = & \emptyset \\ [\![1]\!](t) & = & \{t\} \\ [\![\alpha]\!](t) & = & \{t \mid \mathsf{pred}(\alpha, t) = \mathsf{true}\} \\ [\![\neg a]\!](t) & = & \{t \mid [\![a]\!](t) = \emptyset\} \\ [\![\pi]\!](t) & = & \{t\langle \sigma', \pi \rangle \mid \sigma' = \mathsf{act}(\pi, \mathsf{last}(t))\} \\ [\![p + q]\!](t) & = & [\![p]\!](t) \cup [\![q]\!](t) \\ [\![p \cdot q]\!](t) & = & ([\![p]\!] \bullet [\![q]\!])(t) \\ [\![p^*]\!](t) & = & \bigcup_{0 \le i} [\![p]\!]^i(t) \end{array}$$

$$\begin{array}{rcl} (f \bullet g)(t) & = & \bigcup_{t' \in f(t)} g(t') \\ f^0(t) = \{t\} & & f^{i+1}(t) = (f \bullet f^i)(t) \\ \mathsf{last}(\ldots \langle \sigma, \_ \rangle) & = & \sigma \end{array}$$

Kleene Algebra axioms

$$\begin{array}{rl} p + (q + r) \equiv (p + q) + r & \text{KA-PLUS-ASSOC} \\ p + q \equiv q + p & \text{KA-PLUS-COMM} \\ p + 0 \equiv p & \text{KA-PLUS-ZERO} \\ p + p \equiv p & \text{KA-PLUS-IDEM} \\ p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r & \text{KA-SEQ-ASSOC} \\ 1 \cdot p \equiv p & \text{KA-SEQ-ONE} \\ p \cdot 1 \equiv p & \text{KA-ONE-SEQ} \\ p \cdot (q + r) \equiv p \cdot q + p \cdot r & \text{KA-DIST-L} \\ (p + q) \cdot r \equiv p \cdot r + q \cdot r & \text{KA-DIST-R} \\ 0 \cdot p \equiv 0 & \text{KA-ZERO-SEQ} \\ p \cdot 0 \equiv 0 & \text{KA-SEQ-ZERO} \\ 1 + p \cdot p^* \equiv p* & \text{KA-UNROLL-L} \\ 1 + p^* \cdot p \equiv p* & \text{KA-UNROLL-R} \\ q + p \cdot r \le r \;\to\; p^* \cdot q \le r & \text{KA-LFP-L} \\ p + q \cdot r \le q \;\to\; p \cdot r^* \le q & \text{KA-LFP-R} \end{array}$$

$$p \le q \Leftrightarrow p + q \equiv q$$

Boolean Algebra axioms

$$\begin{array}{rl} a + (b \cdot c) \equiv (a + b) \cdot (a + c) & \text{BA-PLUS-DIST} \\ a + 1 \equiv 1 & \text{BA-PLUS-ONE} \\ a + \neg a \equiv 1 & \text{BA-EXCL-MID} \\ a \cdot b \equiv b \cdot a & \text{BA-SEQ-COMM} \\ a \cdot \neg a \equiv 0 & \text{BA-CONTRA} \\ a \cdot a \equiv a & \text{BA-SEQ-IDEM} \end{array}$$

Consequences

$$\begin{array}{rl} p \cdot a \equiv b \cdot p \;\to\; p \cdot \neg a \equiv \neg b \cdot p & \text{PUSHBACK-NEG} \\ p \cdot (q \cdot p)^* \equiv (p \cdot q)^* \cdot p & \text{SLIDING} \\ (p + q)^* \equiv p^* \cdot (q \cdot p^*)^* & \text{DENESTING} \\ p \cdot a \equiv a \cdot q + r \to & \\ p^* \cdot a \equiv (a + p^* \cdot r) \cdot q^* & \text{STAR-INV} \\ p \cdot a \equiv a \cdot q + r \to & \\ p \cdot a \cdot (p \cdot a)^* \equiv (a \cdot q + r) \cdot (q + r)^* & \text{STAR-EXPAND} \end{array}$$

Fig. 3. Semantics and equational theory for $\mathcal{T}^*$

semantics, though we have skipped the formal detail here. This way we can define composite primitive predicates as in, e.g., temporal logic (Sec. 3.6).

The client's act function takes a primitive action π and the last state in the trace, returning a new state. Whatever new state comes out is recorded in the trace, along with the action just performed.

2.2 Soundness

Proving that the equational theory is sound is relatively straightforward: we only depend on the client's act and pred functions, and none of our KAT axioms (Fig. 3) even mention the client's primitives. We believe the pushback negation theorem (PUSHBACK-NEG) is novel (though it holds in all KATs). Our soundness proof naturally enough requires that any equations the client theory adds need to be sound in our trace semantics. We do need to use several KAT theorems in our completeness proof (Fig. 3, Consequences), the most complex being star expansion (STAR-EXPAND), which we take from Temporal NetKAT [8]; we believe PUSHBACK-NEG is a novel theorem that holds in all KATs.

THEOREM 2.1 (SOUNDNESS OF $\mathcal{T}^*$). *If $\mathcal{T}$'s equational reasoning is sound ($p \equiv_{\mathcal{T}} q \Rightarrow [\![p]\!] = [\![q]\!]$) then $\mathcal{T}^*$'s equational reasoning is sound ($p \equiv q \Rightarrow [\![p]\!] = [\![q]\!]$).*

PROOF. By induction on the derivation of $p \equiv q$.[1] □

2.3 Normalization via pushback

In order to prove completeness (Sec. 2.4), we reduce our KAT terms to a more manageable subset of *normal forms*. Normalization happens via a generalization of weakest preconditions; we use a *pushback* operation to translate a term p into an equivalent term of the form $\sum a_i \cdot m_i$ where each m_i does not contain any tests. Once in this form, we can use the completeness result provided by the client theory to reduce the completeness of our language to an existing result for Kleene algebra.

In order to use our general normalization procedure, the client theory $\mathcal{T}$ must define two things:

(1) a way to extract subterms from predicates, to define an ordering on predicates that serves as the termination measure on normalization (Sec. 2.3.1); and
(2) weakest preconditions for primitives (Sec. 2.3.2).

Once we've defined our normalization procedure, we can use it prove completeness (Sec. 2.4).

2.3.1 Normalization and the maximal subterm ordering. Our normalization algorithm works by "pushing back" predicates to the front of a term until we reach a normal form with *all* predicates at the front. The pushback algorithm's termination measure is a complex one. For example, pushing a predicate back may not eliminate the predicate even though progress was made in getting predicates to the front. More trickily, it may be that pushing test a back through π yields $\sum a_i \cdot \pi$ where each of the a_i is a copy of some subterm of a—and there may be *many* such copies!

Let the set of *restricted actions* $\mathcal{T}_{\mathsf{RA}}$ be the subset of $\mathcal{T}^*$ where the only test is 1. We will use metavariables m, n, and l to denote elements of $\mathcal{T}_{\mathsf{RA}}$. Let the set of *normal forms* $\mathcal{T}^*_{\mathsf{nf}}$ be a set of pairs of tests $a_i \in \mathcal{T}^*_{\mathsf{pred}}$ and restricted actions $m_i \in \mathcal{T}_{\mathsf{RA}}$. We will use metavariables t, u, v, w, x, y, and z to denote elements of $\mathcal{T}^*_{\mathsf{nf}}$; we typically write these sets not in set notation, but as sums, i.e., $x = \sum_{i=1}^{k} a_i \cdot m_i$ means $x = \{(a_1, m_1), (a_2, m_2), \ldots, (a_k, m_k)\}$. The sum notation is convenient, but it is important that normal forms really be treated as sets—there should be no duplicated terms in the sum. We write $\sum_i a_i$ to denote the normal form $\sum_i a_i \cdot 1$. The set of normal forms, $\mathcal{T}^*_{\mathsf{nf}}$, is closed over parallel composition by simply joining the sums. The fundamental challenge in our normalization method is to define sequential composition and Kleene star on $\mathcal{T}^*_{\mathsf{nf}}$.

Our normalization algorithm uses the *maximal subterm ordering* as its termination measure. Due to space constraints, we provide the formal definitions of maximal tests and subterms in the supplemental material. Here we simply give intuition for the two relevant high-level operations: $\mathsf{mt}(x) \subseteq \mathcal{T}^*_{\mathsf{pred}}$ computes the *maximal tests* of a normal form x, which are those tests that are not subterms of any other test; the maximal subterm ordering $x \preceq y$ for normal forms holds when the subterms of x's maximal tests are a subset of the subterms of y's maximal tests. Informally, we have $x \preceq y$ when every test in x is somehow "covered" by a test in y; we have $x \prec y$ when $x \preceq y$ and y has some maximal test x that does not. Our definition of subterms needs the client theory to identify the subterms of its primitives via a function $\mathsf{sub}_{\mathcal{T}}$ such that (1) if $b \in \mathsf{sub}_{\mathcal{T}}(a)$ then $\mathsf{sub}(b) \subseteq \mathsf{sub}_{\mathcal{T}}(a)$ and (2) if $b \in \mathsf{sub}_{\mathcal{T}}(a)$, then either $b \in \{0, 1, a\}$ or b precedes a in a global well ordering of predicates.

LEMMA 2.2 (SPLITTING). *If $a \in \mathsf{mt}(x)$, then there exist y and z such that $x \equiv a \cdot y + z$ and $y \prec x$ and $z \prec x$.*

[1]Full proofs with all necessary lemmas are available in an extended version of this paper in the supplementary material.

Splitting is the key lemma for making progress pushing tests back, allowing us to take a normal form and slowly push its maximal tests to the front; its proof follows from a chain of lemmas given in the supplementary material.

2.3.2 Pushback. In order to define normalization—necessary for completeness (Sec. 2.4)—the client theory must have a *weakest preconditions* operation that respects the subterm ordering.

Definition 2.3 (Weakest preconditions). The *weakest precondition* operation of the client theory is a relation $\mathsf{WP} \subseteq \mathcal{T}_\pi \times \mathcal{T}_\alpha \times \mathcal{P}(\mathcal{T}^*_{\mathrm{pred}})$, where $\mathcal{T}_\pi$ are the primitive actions and $\mathcal{T}_\alpha$ are the primitive predicates of $\mathcal{T}$. We write the relation as $\pi \cdot \alpha \;\mathsf{WP}\; \sum a_i \cdot \pi$ and read it as "α pushes back through π to yield $\sum a_i \cdot \pi$"; the second π is redundant but written for clarity. We require that if $\pi \cdot \alpha \;\mathsf{WP}\; \{a_1, \ldots, a_k\} \cdot \pi$, then $\pi \cdot \alpha \equiv \sum_{i=1}^{k} a_i \cdot \pi$, and $a_i \preceq \alpha$.

Given the client theory's weakest-precondition relation WP, we define a normalization procedure for $\mathcal{T}^*$ by extending the client's WP relation to a more general *pushback* relation, PB (Fig. 4). The client's WP relation need not be a function, nor do the a_i need to be obviously related to α or π in any way. Even when the WP relation is a function, the PB relation will generally not be a function. While WP computes the classical weakest precondition, the PB relations do something different: when pushing back we have the freedom to *change the program itself*—not normally an option for weakest preconditions (see Sec. 7).

We define the top-level normalization routine with the $p \;\mathsf{norm}\; x$ relation (Fig. 4), a syntax directed relation that takes a term p and produces a normal form $x = \sum_i a_i m_i$. Most syntactic forms are easy to normalize: predicates are already normal forms (Pred); primitive actions π are normal forms where there's just one summand and the predicate is 1 (Act); and parallel composition of two normal forms means just joining the sums (Par). But sequence and Kleene star are harder: we define judgments using PB to lift these operations to normal forms (Seq, Star).

For sequences, we can recursively take $p \cdot q$ and normalize p into $x = \sum a_i \cdot m_i$ and q into $y = \sum b_j \cdot n_j$. But how can we combine x and y into a new normal form? We can concatenate and rearrange the normal forms to get $\sum_{i,j} a_i \cdot m_i \cdot b_j \cdot n_j$. If we can push b_j back through m_i to find some new normal form $\sum c_k \cdot l_k$, then $\sum_{i,j,k} a_i \cdot c_k \cdot l_k \cdot n_j$ is a normal form (Join); we write $x \cdot y \;\mathsf{PB}^{\mathsf{J}}\; z$ to mean that the concatenation of x and y is equivalent to the normal form z—the $\cdot$ is suggestive notation.

For Kleene star, we can take p^* and normalize p into $x = \sum a_i \cdot m_i$, but x^* isn't a normal form—we need to somehow move all of the tests out of the star and to the front. We do so with the PB^* relation (Fig. 4), writing $x^* \;\mathsf{PB}^*\; y$ to mean that the Kleene star of x is equivalent to the normal form y—the * on the left is again suggestive notation. The PB^* relation is more subtle than PB^{J}. There are four possible ways to treat x, based on how it splits (Lemma 2.2): if $x = 0$, then our work is trivial since $0^* \equiv 1$ (StarZero); if x splits into $a \cdot x'$ where a is a maximal test and there are no other summands, then we can either use the KAT sliding lemma to pull the test out when a is strictly the largest test in x (Slide) or by using the KAT expansion lemma (Expand); if x splits into $a \cdot x' + z$, we use the KAT denesting lemma to pull a out before recurring on what remains (Denest).

The bulk of the pushback's work happens in the $\mathsf{PB}^\bullet$ relation, which pushes a test back through a restricted action; PB^{R} and PB^{T} use $\mathsf{PB}^\bullet$ to push tests back through normal forms and normal forms back through restricted actions, respectively. To handle negation, the function nnf translates predicates to *negation normal form*, where negations only appear on primitive predicates (Fig. 4); Pushback-Neg justifies this case.

We show that our notion of pushback is correct in two steps. First we prove that pushback is partially correct, i.e., if we can form a derivation in the pushback relations, the right-hand sides are equivalent to the left-hand-sides (Theorem 2.4). Once we've established that our pushback

Normalization $\boxed{p \text{ norm } x}$

$$\frac{}{a \text{ norm } a}\ \textsc{Pred} \qquad \frac{}{\pi \text{ norm } 1 \cdot \pi}\ \textsc{Act} \qquad \frac{p \text{ norm } x \quad q \text{ norm } y}{p + q \text{ norm } x + y}\ \textsc{Par}$$

$$\frac{p \text{ norm } x \quad q \text{ norm } y \quad x \cdot y \text{ PB}^{\text{J}}\ z}{p \cdot q \text{ norm } z}\ \textsc{Seq} \qquad \frac{p \text{ norm } x \quad x^* \text{ PB}^*\ y}{p^* \text{ norm } y}\ \textsc{Star}$$

Sequential composition of normal forms $\boxed{x \cdot y \text{ PB}^{\text{J}}\ z}$

$$\frac{m_i \cdot b_j \text{ PB}^\bullet\ x_{ij}}{(\sum_i a_i \cdot m_i) \cdot (\sum_j b_j \cdot n_j) \text{ PB}^{\text{J}}\ \sum_i \sum_j a_i \cdot x_{ij} \cdot n_j}\ \textsc{Join}$$

Normalization of star $\boxed{x^* \text{ PB}^*\ y}$

$$\frac{}{0^* \text{ PB}^*\ 1}\ \textsc{StarZero} \qquad \frac{x \prec a \quad x \cdot a \text{ PB}^{\text{T}}\ y \quad y^* \text{ PB}^*\ y' \quad y' \cdot x \text{ PB}^{\text{J}}\ z}{(a \cdot x)^* \text{ PB}^*\ 1 + a \cdot z}\ \textsc{Slide}$$

$$\frac{\begin{array}{c} x \not\prec a \quad x \cdot a \text{ PB}^{\text{T}}\ a \cdot t + u \\ (t+u)^* \text{ PB}^*\ y \quad y \cdot x \text{ PB}^{\text{J}}\ z \end{array}}{(a \cdot x)^* \text{ PB}^*\ 1 + a \cdot z}\ \textsc{Expand} \qquad \frac{\begin{array}{c} a \notin \text{mt}(z) \quad y \not\equiv 0 \quad y^* \text{ PB}^*\ y' \\ x \cdot y' \text{ PB}^{\text{J}}\ x' \quad (a \cdot x')^* \text{ PB}^*\ z \quad y' \cdot z \text{ PB}^{\text{J}}\ z' \end{array}}{(a \cdot x + y)^* \text{ PB}^*\ z'}\ \textsc{Denest}$$

Pushback $\boxed{m \cdot a \text{ PB}^\bullet\ y}$ $\boxed{m \cdot x \text{ PB}^{\text{R}}\ y}$ $\boxed{x \cdot a \text{ PB}^{\text{T}}\ y}$

$$\frac{}{m \cdot 0 \text{ PB}^\bullet\ 0}\ \textsc{SeqZero} \qquad \frac{}{m \cdot 1 \text{ PB}^\bullet\ 1 \cdot m}\ \textsc{SeqOne}$$

$$\frac{m \cdot a \text{ PB}^\bullet\ y \quad y \cdot b \text{ PB}^{\text{T}}\ z}{m \cdot (a \cdot b) \text{ PB}^\bullet\ z}\ \textsc{SeqSeqTest} \qquad \frac{n \cdot a \text{ PB}^\bullet\ x \quad m \cdot x \text{ PB}^{\text{R}}\ y}{(m \cdot n) \cdot a \text{ PB}^\bullet\ y}\ \textsc{SeqSeqAction}$$

$$\frac{m \cdot a \text{ PB}^\bullet\ x \quad m \cdot b \text{ PB}^\bullet\ y}{m \cdot (a+b) \text{ PB}^\bullet\ x + y}\ \textsc{SeqParTest} \qquad \frac{m \cdot a \text{ PB}^\bullet\ x \quad n \cdot a \text{ PB}^\bullet\ y}{(m+n) \cdot a \text{ PB}^\bullet\ x + y}\ \textsc{SeqParAction}$$

$$\frac{\pi \cdot \alpha \text{ WP } \{a_1, \dots\}}{\pi \cdot \alpha \text{ PB}^\bullet\ \sum_i a_i \cdot \pi}\ \textsc{Prim} \qquad \frac{\pi \cdot a \text{ PB}^\bullet\ \sum_i a_i \cdot \pi \quad \text{nnf}(\neg(\sum_i a_i)) = b}{\pi \cdot \neg a \text{ PB}^\bullet\ b \cdot \pi}\ \textsc{PrimNeg}$$

$$\frac{\begin{array}{c} m \cdot a \text{ PB}^\bullet\ x \quad x \prec a \\ m^* \cdot x \text{ PB}^{\text{R}}\ y \end{array}}{m^* \cdot a \text{ PB}^\bullet\ a + y}\ \textsc{SeqStarSmaller} \qquad \frac{\begin{array}{c} m \cdot a \text{ PB}^\bullet\ a \cdot t + u \quad m^* \cdot u \text{ PB}^{\text{R}}\ x \\ t^* \text{ PB}^*\ y \quad x \cdot y \text{ PB}^{\text{J}}\ z \end{array}}{m^* \cdot a \text{ PB}^\bullet\ a \cdot y + z}\ \textsc{SeqStarInv}$$

$$\frac{m \cdot a_i \text{ PB}^\bullet\ x_i}{m \cdot \sum_i a_i \cdot n_i \text{ PB}^{\text{R}}\ \sum_i x_i \cdot n_i}\ \textsc{Restricted} \qquad \frac{m_i \cdot a \text{ PB}^\bullet\ \sum_j b_{ij} \cdot m_{ij}}{(\sum_i a_i \cdot m_i) \cdot a \text{ PB}^{\text{T}}\ \sum_i \sum_j a_i \cdot b_{ij} \cdot m_{ij}}\ \textsc{Test}$$

Negation normal form $\boxed{\text{nnf} : \mathcal{T}^*_{\text{pred}} \to \mathcal{T}^*_{\text{pred}}}$

$$\begin{array}{rcl} \text{nnf}(0) &=& 0 \\ \text{nnf}(1) &=& 1 \\ \text{nnf}(\alpha) &=& \alpha \\ \text{nnf}(a+b) &=& \text{nnf}(a) + \text{nnf}(b) \\ \text{nnf}(a \cdot b) &=& \text{nnf}(a) \cdot \text{nnf}(b) \end{array} \qquad \begin{array}{rcl} \text{nnf}(\neg 0) &=& 1 \\ \text{nnf}(\neg 1) &=& 0 \\ \text{nnf}(\neg \alpha) &=& \neg \alpha \\ \text{nnf}(\neg\neg a) &=& \text{nnf}(a) \\ \text{nnf}(\neg(a+b)) &=& \text{nnf}(\neg a) \cdot \text{nnf}(\neg b) \\ \text{nnf}(\neg(a \cdot b)) &=& \text{nnf}(\neg a) + \text{nnf}(\neg b) \end{array}$$

Fig. 4. Normalization for $\mathcal{T}^*$

relations' derivations mean what we want, we have to show that we can find such derivations; here we use our maximal subterm measure to show that the recursive tangle of our PB relations always terminates (Theorem 2.5) .

Theorem 2.4 (Pushback soundness). *For each of the* PB *relations, the left side is equivalent to the right side, e.g., if* x^* PB* y *then* $x^* \equiv y$.

Proof. By simultaneous induction on the derivations. Most cases follow by the IH and axioms, with a few relying on KAT theorems like sliding, denesting, star expansion [8], and pushback negation (Fig. 3, Consequences). □

Theorem 2.5 (Pushback existence). *For each of the* PB *relations, every left side relates to a right side that is no larger, e.g., for all* x *there exists* $y \leq x$ *such that* x^* PB* y.

Proof. By induction on the lexicographical order of: the subterm ordering ($\prec$); the size of x; the size of m; and the size of a. Cases go by using splitting (Lemma 2.2) to show that derivations exist followed by subterm ordering congruence to find orderings to apply the IH. □

Finally, to reiterate our discussion of PB$^\bullet$, Theorem 2.5 shows that every left-hand side of the pushback relation has a corresponding right-hand side. We *haven't* proved that the pushback relation is functional— if a term has more than one maximal test, there could be many different choices of how we perform the pushback.

Now that we can push back tests, we can show that every term has an equivalent normal form.

Corollary 2.6 (Normal forms). *For all* $p \in \mathcal{T}^*$, *there exists a normal form* x *such* p norm x *and that* $p \equiv x$.

Proof. By induction on p, using Theorems 2.5 and 2.4 in the Seq and Star case. □

The PB relations and these two proofs are one of the contributions of this paper: we believe it is the first time that a KAT normalization procedure has been made so explicit, rather than hiding inside of completeness proofs. Temporal NetKAT, which introduced the idea of pushback, proved a concretization of Theorems 2.4 and 2.5 as a single theorem and without any explicit normalization or pushback relation.

2.4 Completeness

We prove completeness—if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$—by normalizing p and q and comparing the resulting terms. Our completeness proof uses the completeness of Kleene algebra (KA) as its foundation: the set of possible traces of actions performed for a restricted (test-free) action in our denotational semantics is a regular language, and so the KA axioms are sound and complete for it. In order to relate our denotational semantics to regular languages, we define the regular interpretation of restricted actions $m \in \mathcal{T}_{\text{RA}}$ in the conventional way and then relate our denotational semantics to the regular interpretation (Fig. 5). Readers familiar with NetKAT's completeness proof may notice that we've omitted the language model and gone straight to the regular interpretation. We're able to shorten our proof because our tracing semantics is more directly relatable to its regular interpretation, and because our completeness proof separately defers to the client theory's decision procedure for the predicates at the front. Our normalization routine—the essence of our proof—only uses the KAT axioms and doesn't rely on any property of our tracing semantics. We conjecture that one could prove a similar completeness result and derive a similar decision procedure with a merging, non-tracing semantics, like in NetKAT or KAT+B! [1, 29]. We haven't attempted the proof or an implementation.

$$
\begin{array}{rcl}
\mathcal{R} & : & \mathcal{T}_{\mathsf{RA}} \to \mathcal{P}(\Pi_{\mathcal{T}}^*) \\
\mathcal{R}(1) & = & \{\epsilon\} \\
\mathcal{R}(\pi) & = & \{\pi\} \\
\mathcal{R}(m+n) & = & \mathcal{R}(m) \cup \mathcal{R}(n) \\
\mathcal{R}(m \cdot n) & = & \{uv \mid u \in \mathcal{R}(m), v \in R(n)\} \\
\mathcal{R}(m^*) & = & \bigcup_{0 \leq i} \mathcal{R}(m)^i
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{label} & : & \mathsf{Trace} \to \Pi_{\mathcal{T}}^* \\
\mathsf{label}(\langle \sigma, \bot \rangle) & = & \epsilon \\
\mathsf{label}(t\langle \sigma, \pi \rangle) & = & \mathsf{label}(t)\pi \\
 & & \\
\mathcal{L}^0 & = & \{\epsilon\} \\
\mathcal{L}^{n+1} & = & \{uv \mid u \in \mathcal{L}, v \in \mathcal{L}^n\}
\end{array}
$$

Fig. 5. Regular interpretation of restricted actions

Lemma 2.7 (Labels are regular). $\{\mathsf{label}([\![m]\!](\langle \sigma, \bot \rangle)) \mid \sigma \in \mathsf{State}\} = \mathcal{R}(m)$

Proof. By induction on the restricted action m. □

Theorem 2.8 (Completeness). *If the emptiness of $\mathcal{T}$ predicates is decidable, then if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$.*

Proof. There must exist normal forms x and y such that p norm x and q norm y and $p \equiv x$ and $q \equiv y$ (Corollary 2.6); by soundness (Theorem 2.1), we can find that $[\![p]\!] = [\![x]\!]$ and $[\![q]\!] = [\![y]\!]$, so it must be the case that $[\![x]\!] = [\![y]\!]$. We will find a proof that $x \equiv y$; we can then transitively construct a proof that $p \equiv q$.

We have $x = \sum_i a_i \cdot m_i$ and $y = \sum_j b_j \cdot n_j$. In principle, we ought to be able to match up each of the a_i with one of the b_j and then check to see whether m_i is equivalent to n_j (by appealing to the completeness on Kleene algebra). But we can't simply do a syntactic matching—we could have a_i and b_j that are in effect equivalent, but not obviously so. Worse still, we could have a_i and $a_{i'}$ equivalent! We need to perform two steps of disambiguation: first each normal form must be unambiguous on its own, and then they must be pairwise unambiguous between the two normal forms.

To construct independently unambiguous normal forms, we explode our normal form x into a disjoint form $\hat{x}$, where we test each possible combination of a_i and run the actions corresponding to the true predicates, i.e., m_i gets run precisely when a_i is true:

$$
\begin{array}{rl}
\hat{x} = & a_1 \cdot a_2 \cdot \dots \cdot a_n \cdot m_1 \cdot m_2 \cdot \dots \cdot m_n \\
+ & \neg a_1 \cdot a_2 \cdot \dots \cdot a_n \cdot m_2 \cdot \dots \cdot m_n \\
+ & a_1 \cdot \neg a_2 \cdot \dots \cdot a_n \cdot m_1 \cdot \dots \cdot m_n \\
+ & \dots \\
+ & \neg a_1 \cdot \neg a_2 \cdot \dots \cdot a_n \cdot m_n
\end{array}
$$

and similarly for $\hat{y}$. We can find $x \equiv \hat{x}$ via distributivity (BA-Plus-Dist) and the excluded middle (BA-Excl-Mid).

Given normal forms with locally disjoint cases, we can take the Cartesian product of $\hat{x}$ and $\hat{y}$, which allows us to do a *syntactic* comparison on each of the predicates. Let $\ddot{x}$ and $\ddot{y}$ be the extension of $\hat{x}$ and $\hat{y}$ with the tests from the other form, giving us $\ddot{x} = \sum_{i,j} c_i \cdot d_j \cdot l_i$ and $\ddot{y} = \sum_{i,j} c_i \cdot d_j \cdot m_j$. Extending the normal forms to be disjoint between the two normal forms is still provably equivalent using commutativity (BA-Seq-Comm), distributivity (BA-Plus-Dist), and the excluded middle (BA-Excl-Mid).

Now that each of the predicates are syntactically uniform and disjoint, we can proceed to compare the commands. But there is one final risk: what if the $c_i \cdot d_j \equiv 0$? Then l_i and o_j could safely be different. We therefore use the client's emptiness checker to eliminate those cases where the expanded tests at the front of $\ddot{x}$ and $\ddot{y}$ are equivalent to zero, which is sound by the client theory's completeness and zero-cancellation (KA-Zero-Seq and KA-Seq-Zero).

Syntax

$$\begin{array}{rcl} \alpha & ::= & b = \text{true} \\ \pi & ::= & b := \text{true} \mid b := \text{false} \\ \text{sub}(\alpha) & = & \{\alpha\} \end{array}$$

Semantics

$$\begin{array}{lcl} b & \in & \mathcal{B} \\ \text{State} & = & \mathcal{B} \to \{\text{true}, \text{false}\} \\ \text{pred}(b = \text{true}, t) & = & \text{last}(t)(b) \\ \text{act}(b := \text{true}, \sigma) & = & \sigma[b \mapsto \text{true}] \\ \text{act}(b := \text{false}, \sigma) & = & \sigma[b \mapsto \text{false}] \end{array}$$

Weakest precondition

$$\begin{array}{l} b := \text{true} \cdot b = \text{true WP } 1 \\ b := \text{false} \cdot b = \text{true WP } 0 \end{array}$$

Axioms

$$\begin{array}{rl} (b := \text{true}) \cdot (b = \text{true}) \equiv (b := \text{true}) & \textsc{Set-Test-True-True} \\ (b := \text{false}) \cdot (b = \text{true}) \equiv 0 & \textsc{Set-Test-False-True} \end{array}$$

Fig. 6. BitVec, theory of bitvectors

Finally, we can defer to deductive completeness for KA to find proofs that the commands are equal. To use KA's completeness to get a proof over commands, we have to show that if our commands have equal denotations in our semantics, then they will also have equal denotations in the KA semantics. We've done exactly this by showing that restricted actions have regular interpretations: because the zero-canceled $\ddot{x}$ and $\ddot{y}$ are provably equal, soundness guarantees that their denotations are equal. Since their tests are pairwise disjoint, if their denotations are equal, it must be that any non-canceled commands are equal, which means that each label of these commands must be equal—and so $\mathcal{R}(l_i) = \mathcal{R}(o_j)$ (Lemma 2.7). By the deductive completeness of KA, we know that KA $\vdash l_i \equiv o_j$. Since we have the KA axioms in our system, then $l_i \equiv o_j$; by reflexivity, we know that $c_i \cdot d_j \equiv c_i \cdot d_j$, and we have proved that $\ddot{x} \equiv \ddot{y}$. By transitivity, we can see that $\hat{x} \equiv \hat{y}$ and so $x \equiv y$ and $p \equiv q$, as desired. □

3 CASE STUDIES

In this section, we define KAT client theories for bitvectors and networks, as well as higher-order theories for products of theories, sets over theories, and temporal logic over theories.

3.1 Bit vectors

The simplest KMT is bit vectors: we extend KAT with some finite number of bits, each of which can be set to true or false and tested for their current value (Fig. 6). The theory adds actions $b := \text{true}$ and $b := \text{false}$ for boolean variables b, and tests of the form $b = \text{true}$, where b is drawn from some set of names $\mathcal{B}$. Since our bit vectors are embedded in a KAT, we can use KAT operators to build up encodings on top of bits: $b = \text{false}$ desugars to $\neg(b = \text{true})$; flip b desugars to $(b = \text{true} \cdot b := \text{false}) + (b = \text{false} \cdot b := \text{true})$. We could go further and define numeric operators on collections of bits, at the cost of producing larger terms. We are not limited to just numbers, of course; once we have bits, we can encode any bounded data structure we like.

KAT+B! [29] develops a nearly identical theory, though our semantics admit different equations. We use a *trace* semantics, where we distinguish between $(b := \text{true} \cdot b := \text{true})$ and $(b := \text{true})$. Even though the final states are equivalent, they produce different traces because they run different actions. KAT+B!, on the other hand, doesn't distinguish based on the trace of actions, so they find that $(b := \text{true} \cdot b := \text{true}) \equiv (b := \text{true})$. It's difficult to say whether one model is better than the other—we imagine that either could be appropriate, depending on the setting. For example, our trace semantics is useful for answering model-checking-like questions (Sec. 3.4).

3.2 Disjoint products

Given two client theories, we can combine them into a disjoint product theory, $\text{Prod}(\mathcal{T}_1, \mathcal{T}_2)$, where the states are products of the two sub-theory's states and the predicates and actions from $\mathcal{T}_1$ can't

Syntax

$$\begin{array}{rcl} \alpha & ::= & \alpha_1 \mid \alpha_2 \\ \pi & ::= & \pi_1 \mid \pi_2 \\ \mathsf{sub}(\alpha_i) & = & \mathsf{sub}_i(\alpha_i) \end{array}$$

Semantics

$$\begin{array}{lcl} \mathsf{State} & = & \mathsf{State}_1 \times \mathsf{State}_2 \\ \mathsf{pred}(\alpha_i, t) & = & \mathsf{pred}_i(\alpha_i, t_i) \\ \mathsf{act}(\pi_i, \sigma) & = & \sigma[\sigma_i \mapsto \mathsf{act}_i(\pi_i, \sigma_i)] \end{array}$$

Weakest precondition extending $\mathcal{T}_1$ and $\mathcal{T}_2$

$$\pi_1 \cdot \alpha_2 \text{ WP } \alpha_2 \qquad \pi_2 \cdot \alpha_1 \text{ WP } \alpha_1$$

Axioms extending $\mathcal{T}_1$ and $\mathcal{T}_2$

$$\begin{array}{ll} \pi_1 \cdot \alpha_2 \equiv \alpha_2 \cdot \pi_1 & \textsc{L-R-Comm} \\ \pi_2 \cdot \alpha_1 \equiv \alpha_1 \cdot \pi_2 & \textsc{R-L-Comm} \end{array}$$

Fig. 7. $\mathsf{Prod}(\mathcal{T}_1, \mathcal{T}_2)$, products of two disjoint theories

Syntax

$$\begin{array}{rcl} \alpha & ::= & \mathsf{in}(x, c) \mid e = c \mid \alpha_e \\ \pi & ::= & \mathsf{add}(x, e) \mid \pi_e \\ \mathsf{sub}(\mathsf{in}(x, c)) & = & \{\mathsf{in}(x, c)\} \cup \mathsf{sub}(\neg(e = c)) \\ \mathsf{sub}(e = c) & = & \mathsf{sub}(e = c) \\ \mathsf{sub}(\alpha_e) & = & \mathsf{sub}(\alpha_e) \end{array}$$

Semantics

$$\begin{array}{ll} c & \in \mathcal{C} \\ e & \in \mathcal{E} \\ x & \in \mathcal{V} \\ \mathsf{State} & = (\mathcal{V} \to \mathcal{P}(\mathcal{C})) \times (\mathcal{E} \to \mathcal{C}) \\ \mathsf{pred}(\mathsf{in}(x, c), t) & = \mathsf{last}(t)_2(c) \in \mathsf{last}(t)_1(x) \\ \mathsf{pred}(\alpha_e, t) & = \mathsf{pred}(\alpha_e, t_2) \\ \mathsf{act}(\mathsf{add}(x, e), \sigma) & = \sigma[\sigma_1[x \mapsto \sigma_1(x) \cup \{\sigma(e)\}]] \\ \mathsf{act}(\pi_e, \sigma) & = \sigma[\sigma_2 \mapsto \mathsf{act}(\pi_e, \sigma_2)] \end{array}$$

Weakest precondition extending $\mathcal{E}$

$$\begin{array}{l} \mathsf{add}(y, e) \cdot \mathsf{in}(x, c) \text{ WP } \mathsf{in}(x, c) \\ \mathsf{add}(x, e) \cdot \mathsf{in}(x, c) \text{ WP } (e = c) + \mathsf{in}(x, c) \\ \mathsf{add}(x, e) \cdot \alpha_e \text{ WP } \alpha_e \end{array}$$

Axioms extending $\mathcal{E}$

$$\begin{array}{rl} \mathsf{add}(y, e) \cdot \mathsf{in}(x, c) \equiv \mathsf{in}(x, c) \cdot \mathsf{add}(y, e) & \textsc{Add-Comm} \\ \mathsf{add}(x, e) \cdot \mathsf{in}(x, c) \equiv ((e = c) + \mathsf{in}(x, c)) \cdot \mathsf{add}(x, e) & \textsc{Add-In} \\ \mathsf{add}(x, e) \cdot \alpha_e \equiv \alpha_e \cdot \mathsf{add}(x, e) & \textsc{Add-Comm2} \end{array}$$

Fig. 8. $\mathsf{Set}(\mathcal{E})$, unbounded sets over expressions

affect $\mathcal{T}_2$ and vice versa (Fig. 7). We explicitly give definitions for pred and act that defer to the corresponding sub-theory, using t_i to project the trace state to the ith component. It may seem that disjoint products don't give us much, but they in fact allow for us to simulate much more interesting languages in our derived KATs. For example, Prod(BitVec, IncNat) allows us to program with both variables valued as either booleans or (increasing) naturals; the product theory lets us directly express the sorts of programs that Kozen's early static analysis work had to encode manually, i.e., loops over boolean and numeric state [32].

3.3 Unbounded sets

We define a KMT for unbounded sets parameterized on a theory of expressions $\mathcal{E}$ (Fig. 8). The set data type supports just one operation: $\mathsf{add}(x, e)$ adds the value of expression e to set x (we could add $\mathsf{del}(x, e)$, but we omit it to save space). It also supports a single test: $\mathsf{in}(x, c)$ checks if the constant c is contained in set x. The idea is that $e \in \mathcal{E}$ refers to expressions with, say, variables x and constants c. We allow arbitrary expressions e in some positions and constants c in others. (If we allowed expressions in all positions, WP wouldn't necessarily be non-increasing.)

To instantiate the Set theory, we need a few things: expressions $\mathcal{E}$, a subset of *constants* $\mathcal{C} \subseteq \mathcal{E}$, and predicates for testing (in)equality between expressions and constants ($e = c$ and $e \neq c$). (We can not, in general, expect tests for equality of non-constant expressions, as it may cause us to accidentally define a counter machine.) We treat these two extra predicates as inputs, and expect that they have well behaved subterms. Our state has two parts: $\sigma_1 : \mathcal{V} \to \mathcal{P}(\mathcal{C})$ records the current sets for each set in $\mathcal{V}$, while $\sigma_2 : \mathcal{E} \to \mathcal{C}$ evaluates expressions in each state. Since each state has its own evaluation function, the expression language can have actions that update σ_2.

Syntax

$$\alpha ::= \bigcirc a \mid a \mathcal{S} b \mid a$$
$$\pi ::= \pi_{\mathcal{T}}$$
$$\begin{aligned} \mathrm{sub}(\bigcirc a) &= \{\bigcirc a\} \cup \mathrm{sub}(a) \\ \mathrm{sub}(a \mathcal{S} b) &= \{a \mathcal{S} b\} \cup \mathrm{sub}(a) \cup \mathrm{sub}(b) \\ \mathrm{act}(\pi, \sigma) &= \mathrm{act}(\pi, \sigma) \end{aligned}$$

$$\bullet a = \neg \bigcirc \neg a \qquad a \mathcal{B} b = a \mathcal{S} b + \Box a$$
$$\mathrm{start} = \neg \bigcirc 1 \qquad \Diamond a = 1 \mathcal{S} a \qquad \Box a = \neg \Diamond \neg a$$

Semantics

$$\begin{aligned} &\mathrm{State} && = \mathrm{State}_{\mathcal{T}} \\ &\mathrm{pred}(\bigcirc a, \langle \sigma, l \rangle) && = \mathrm{false} \\ &\mathrm{pred}(\bigcirc a, t\langle \sigma, l \rangle) && = \mathrm{pred}(a, t) \\ &\mathrm{pred}(a \mathcal{S} b, \langle \sigma, l \rangle) && = \mathrm{pred}(b, \langle \sigma, l \rangle) \\ &\mathrm{pred}(a \mathcal{S} b, t\langle \sigma, l \rangle) && = \mathrm{pred}(b, t\langle \sigma, l \rangle) \lor \\ &&& \quad (\mathrm{pred}(a, t\langle \sigma, l \rangle) \land \mathrm{pred}(a \mathcal{S} b, t)) \end{aligned}$$

Weakest precondition extending $\mathcal{T}$

$$\pi \cdot \bigcirc a \text{ WP } a$$

$$\frac{\pi \cdot a \text{ PB}^{\bullet}_{\mathcal{T}}\ a' \cdot \pi \qquad \pi \cdot b \text{ PB}^{\bullet}_{\mathcal{T}}\ b' \cdot \pi}{\pi \cdot (a \mathcal{S} b) \text{ WP } b' + a' \cdot (a \mathcal{S} b)}$$

Axioms extending $\mathcal{T}$

inherited from $\mathcal{T}$	
$\bigcirc(a \cdot b) \equiv \bigcirc a \cdot \bigcirc b$	LTL-Last-Dist-Seq
$\bigcirc(a + b) \equiv \bigcirc a + \bigcirc b$	LTL-Last-Dist-Plus
$\bullet 1 \equiv 1$	LTL-WLast-One
$a \mathcal{S} b \equiv b + a \cdot \bigcirc(a \mathcal{S} b)$	LTL-Since-Unroll
$\neg(a \mathcal{S} b) \equiv (\neg b) \mathcal{B} (\neg a \cdot \neg b)$	LTL-Not-Since
$a \leq \bullet a \cdot b \ \rightarrow\ a \leq \Box b$	LTL-Induction
$\Box a \leq \Diamond(\mathrm{start} \cdot a)$	LTL-Finite

Fig. 9. $\mathrm{LTL}_f(\mathcal{T})$, linear temporal logic on finite traces over an arbitrary theory

For example, we can have sets of naturals by setting $\mathcal{E} ::= n \in \mathbb{N} \mid i \in \mathcal{V}'$, where our constants $C = \mathbb{N}$ and $\mathcal{V}'$ is some set of variables distinct from those we use for sets. We can update the variables in $\mathcal{V}'$ using IncNat's actions while simultaneously using set actions to keep sets of naturals. Our KMT can then prove that the term $(\mathrm{inc}_i \cdot \mathrm{add}(x, i))^* \cdot (i > 100) \cdot \mathrm{in}(x, 100)$ is non-empty by pushing tests back (and unrolling the loop 100 times). The set theory's sub function calls the client theory's sub function, so all $\mathrm{in}(x, e)$ formulae must come *later* in the global well ordering than any of those generated by the client theory's $e = c$ or $e \neq c$.

3.4 Past-time linear temporal logic

Past-time linear temporal logic on finite traces (LTL_f) is a *higher-order theory*: LTL_f is itself parameterized on a theory $\mathcal{T}$, which introduces its own predicates and actions—any $\mathcal{T}$ test can appear inside of LTL_f's predicates (Fig. 9). For information on LTL_f, we refer the reader to work by Baier and McIlraith, De Giacomo and Vardi, Roşu, and Beckett et al., and Campbell and Greenberg [5, 8, 10, 11, 17, 18, 45].

LTL_f adds just two predicates: $\bigcirc a$, pronounced "last a", means a held in the prior state; and $a \mathcal{S} b$, pronounced "a since b", means b held at some point in the past, and a has held since then. There is a slight subtlety around the beginning of time: we say that $\bigcirc a$ is false at the beginning (what can be true in a state that never happened?), and $a \mathcal{S} b$ degenerates to b at the beginning of time. The last and since predicates together are enough to encode the rest of LTL_f; encodings are given below the syntax. Weakest preconditions uses inference rules: to push back $\mathcal{S}$, we unroll $a \mathcal{S} b$ into $a \cdot \bigcirc(a \mathcal{S} b) + b$; pushing last through an action is easy, but pushing back a or b recursively uses the $\text{PB}^{\bullet}$ judgment. Adding these rules leaves our judgments monotonic, and if $\pi \cdot a \text{ PB}^{\bullet}\ x$, then $x = \sum a_i \pi$. In this case, our implementation's recursive modules are critical—they allow us to use the derived pushback inside our definition of weakest preconditions.

The equivalence axioms come from Temporal NetKAT [8]; the deductive completeness result for these axioms comes from Campbell and Greenberg's work, which proves deductive completeness

Syntax

$$
\begin{array}{rcl}
\alpha & ::= & f = v \\
\pi & ::= & f \leftarrow v \\
\mathsf{sub}(\alpha) & = & \{\alpha\}
\end{array}
$$

Semantics

$$
\begin{array}{lcl}
\mathsf{F} & = & \text{packet fields} \\
\mathsf{V} & = & \text{packet field values} \\
\mathsf{State} & = & \mathsf{F} \to \mathsf{V} \\
\mathsf{pred}(f = v, t) & = & \mathsf{last}(t).f = v \\
\mathsf{act}(f \leftarrow v, \sigma) & = & \sigma[f \mapsto v]
\end{array}
$$

Weakest precondition

$$
\begin{array}{l}
f \leftarrow v \cdot f = v \text{ WP } 1 \\
f \leftarrow v \cdot f = v' \text{ WP } 0 \text{ when } v \neq v' \\
f' \leftarrow v \cdot f = v \text{ WP } f = v
\end{array}
$$

Axioms

$$
\begin{array}{rl}
f \leftarrow v \cdot f' = v' \equiv f' = v' \cdot f \leftarrow v & \textsc{PA-Mod-Comm} \\
f \leftarrow v \cdot f = v \equiv f \leftarrow v & \textsc{PA-Mod-Filter} \\
f = v \cdot f = v' \equiv 0, \text{ if } v \neq v' & \textsc{PA-Contra} \\
\sum_v f = v \equiv 1 & \textsc{PA-Match-All}
\end{array}
$$

Fig. 10. Tracing NetKAT a/k/a NetKAT without dup

for an axiomatic framing and then relates those axioms to our equations [10, 11]; we could have also used Roşu's proof with coinductive axioms [45].

As a use of LTL_f, recall the simple While program from Sec. 1. We may want to check that, before the last state after the loop, the variable $\mathtt{j}$ was always less than or equal to 200. We can capture this with the test $\bigcirc\Box(\mathtt{j} \leq 200)$. We can use the LTL_f axioms to push tests back through actions; for example, we can rewrite terms using these LTL_f axioms alongside the natural number axioms:

$$
\begin{aligned}
\mathtt{j} := \mathtt{j} + 2 \cdot \Box(\mathtt{j} \leq 200) &\equiv \mathtt{j} := \mathtt{j} + 2 \cdot (\mathtt{j} \leq 200 \cdot \bigcirc\Box(\mathtt{j} \leq 200)) \\
&\equiv (\mathtt{j} := \mathtt{j} + 2 \cdot \mathtt{j} \leq 200) \cdot \bigcirc\Box(\mathtt{j} \leq 200) \\
&\equiv (\mathtt{j} \leq 198) \cdot \mathtt{j} := \mathtt{j} + 2 \cdot \bigcirc\Box(\mathtt{j} \leq 200) \\
&\equiv (\mathtt{j} \leq 198) \cdot \Box(\mathtt{j} \leq 200) \cdot \mathtt{j} := \mathtt{j} + 2
\end{aligned}
$$

Pushing the temporal test back through the action reveals that $\mathtt{j}$ is never greater than 200 if before the action $\mathtt{j}$ was not greater than 198 in the previous state and $\mathtt{j}$ never exceeded 200 before the action as well. The final pushed back test $(j \leq 198) \cdot \Box(j \leq 200)$ satisfies the theory requirements for pushback not yielding larger tests, since the resulting test is only in terms of the original test and its subterms. Note that we've embedded our theory of naturals into LTL_f: we can generate a complete equational theory for LTL_f over any other complete theory.

The ability to use temporal logic in KAT means that we can model check programs by phrasing model checking questions in terms of program equivalence. For example, for some program r, we can check if $r \equiv r \cdot \bigcirc\Box(j \leq 200)$. In other words, if there exists some program trace that does not satisfy the test, then it will be filtered—resulting in non-equivalent terms. If the terms are equal, then every trace from r satisfies the test. Similarly, we can test whether $r \cdot \bigcirc\Box(j \leq 200)$ is empty—if so, there are *no* satisfying traces.

In addition to model checking, temporal logic is a useful programming language feature: programs can make dynamic program decisions based on the past more concisely. Such a feature is useful for Temporal NetKAT (Sec. 3.6 below), but could also be used for, e.g., regular expressions with lookbehind or even a limited form of back-reference.

3.5 Tracing NetKAT

We define NetKAT as a KMT over packets, which we model as functions from packet fields to values (Fig. 10). KMT's trace semantics diverge slightly from NetKAT's: like KAT+B! (Sec. 3.1; [29]), NetKAT normally merges adjacent writes. If the policy analysis demands reasoning about the history of packets traversing the network—reasoning, for example, about which routes packets actually take—the programmer must insert dups to record relevant moments in time. From our

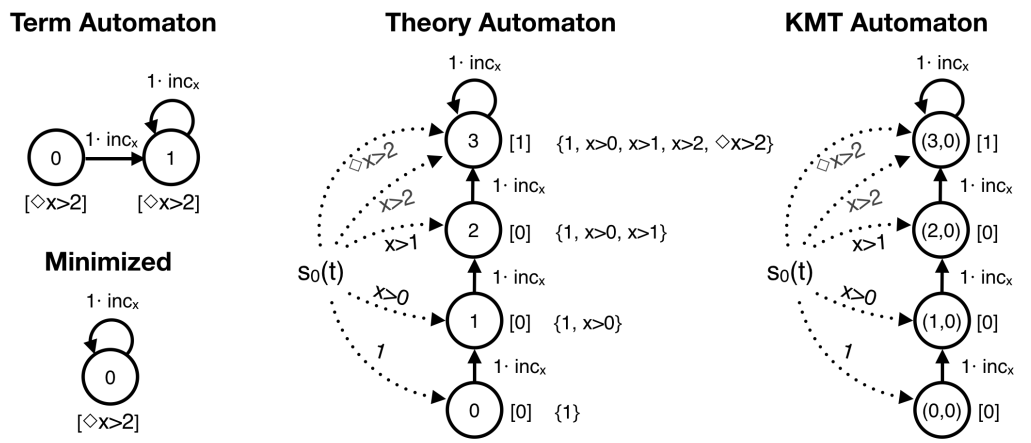

Fig. 11. Automata construction for $\mathtt{inc}_x{}^* \cdot \Diamond x > 2$ in the theory of $\mathrm{LTL}_f(\mathrm{IncNat})$.

perspective, NetKAT very nearly has a tracing semantics, but the traces are selective. If we put an implicit dup before *every* field update, NetKAT has our tracing semantics.

3.6 Temporal NetKAT

We derive Temporal NetKAT as $\mathrm{LTL}_f(\mathrm{NetKAT})$, i.e., LTL_f instantiated over tracing NetKAT; the combination yields precisely the system described in the Temporal NetKAT paper [8]. Our LTL_f theory can now rely on Campbell and Greenberg's proof of deductive completeness for LTL_f [10, 11], we can automatically derive a stronger completeness result for Temporal NetKAT than that from the paper, which showed completeness only for "network-wide" policies, i.e., those with start at the front.

4 AUTOMATA

While the deductive completeness proof (Theorem 2.8 in Sec. 2) gives a way to determine equivalence of KAT terms through normalization, using such rewriting-based proofs as the basis of a decision procedure isn't always practical. But just as pushback yields a novel completeness proof, it can also help provide an automata-theoretic account of equivalence. We compare performance in Sec. 6.

Our automata theory is heavily based on previous work on Antimirov partial derivatives [3] and NetKAT's compiler [51]. We diverge their approach to account for client theory predicates that depend on more than the last state of the trace. Our solution is adapted from the Temporal NetKAT compiler [8]: to construct an automaton for a term in a KMT, we build *two* automata—one for the policy fragment of the term and one for each predicate that occurs therein—and combine the two in a specialized quasi-intersection operation.

A *KMT automaton* is a 4-tuple $(S, s_0, \epsilon, \delta)$, where: the set of automata states S identifies non-initial states (unrelated to State, the state space of the client theory); the *initial state selector* s_0 is a function that takes a trace and selects an initial state; the *acceptance function* $\epsilon : S \times \mathrm{Trace} \to \mathcal{P}(\mathrm{State})$ is a function identifying which theory states (in State) are accepted in each automaton state $s \in S$; the *transition function* $\delta : S \times \mathrm{Trace} \to \mathcal{P}(\mathrm{Log} \times S)$ identifies successor states given an automaton and a single KMT state. Intuitively, the automata works on traces, i.e., sequences of log entries: $\langle \sigma_0, \pi_1 \rangle \ldots \langle \sigma_n, \pi_n \rangle$. While the acceptance and transition functions look at traces, that is an artifact of their construction: they will only actually look at the last state of the input.

Consider the KMT automaton (Fig. 11, rightmost) for the term $\mathsf{inc}_x{}^* \cdot \Diamond x > 2$ taken from the LTL$_f$(IncNat) theory. The automaton accepts a trace of the form: $\langle [x \mapsto 1, \bot] \rangle \langle [x \mapsto 2, \mathsf{inc}_x] \rangle \langle [x \mapsto 3], \mathsf{inc}_x \rangle$. Informally, the initial state selector s_0 looks at the trace so far to determine where to begin a run. In our example, the state (0,0) is used for a trace where x has never been greater than 2 and x is currently 0; we would start in state (1,0) if x were 1. From state (1,0), the automaton will move to state (2,1) and then (3,1) unconditionally for the inc_x action, which corresponds to actions in the log entries of the trace. The acceptance function, written in brackets alongside each state, assigns state (3,1) the condition 1, meaning that all theory states are accepted; no other states are accepting, i.e., their acceptance condition is 0.

The transition function δ takes an automaton state S and a KMT trace and maps them to a set of new pairs of automaton state and and KMT log items (a KMT state/action pair). In the figure, we draw transitions as arcs between states with a pair of a KMT test and a primitive KMT action. For example, the transition from state (1,0) to (2,0) is captured by the term $1 \cdot \mathsf{inc}_x$, i.e., the transition can always fire and increments the value of x.

Taken all together, our KMT automaton captures the fact that there are 4 interesting cases for the term $\mathsf{inc}_x{}^* \cdot \Diamond x > 2$. If the program trace already had $x > 2$ at some point in the past or has $x > 2$ in the current state, then we move to state (3,0) and will accept the trace regardless of how many increment commands are executed in the future. If the initial trace has $x > 1$, then we move to state (2,0). If we see at least one more increment command, then we move to state (3,0) where the trace will be accepted no matter what. If the initial trace has $x > 0$, we move to state (1,0) where we must see at least 2 more increment commands before accepting the trace. Finally, if the initial trace has any other value (here, only $x = 0$ is possible), then we move to state (0,0) and must see at least 3 increment commands before accepting.

4.1 Constructing KMT automata

The KMT automaton for a given term p is constructed in two phases: we first construct a *term automaton* for a version of p where predicates are placed as transition and acceptance conditions. Such a symbolic automaton can be unwieldy—for example, the term automaton in (Fig. 11, top left) has a temporal predicate as an acceptance condition, which is challenging to reason about. We therefore find every predicate mentioned in the term automaton and construct a corresponding *theory automaton* (Fig. 11, middle), using pushback to move tests to the front of the automaton. We finally combine these two to form a KMT automaton with simple acceptance conditions (0 or 1).

4.1.1 Term automata.

The term automaton uses the Antimirov-derivative approach from the NetKAT compiler to construct an automaton for a given term. At this stage, we leave arbitrary predicates on the edges—we use theory automata (Sec. 4.1.2) to resolve those predicates. Formally, our automaton $\mathcal{A}_\pi(p)$ is defined in as a 4-tuple $(S, s_0, \epsilon, \delta)$, where S is a set of states, s_0 is an initial state, ϵ is an acceptance condition, and δ is a transition relation (Fig. 12). The automata's runs are described by the accepts relation, where we say $\mathcal{A}_\pi(p), \ell$ accepts $t; t'$ when the automaton $\mathcal{A}_\pi(p)$ in state ℓ accepts the trace t' after having already seen the trace t. The semi-colon on the right-hand side of the accepts relation can be thought of as a 'cursor' indicating where we are in the trace so far. The NetKAT compiler's automaton doesn't bother keeping the trace, but our predicates can reflect on the entire trace—so we must be careful to keep track of it.

Given a KMT term p, we start constructing the term automaton $\mathcal{A}_\pi(p)$ by annotating each occurrence of each theory action π in p with a unique label ℓ; these labels will form the states of $\mathcal{A}_\pi(p)$. Then we take the partial derivative of p by computing $\mathcal{D}(p)$ (Fig. 12). The derivative computes a set of *linear forms*—tuples of the form $\langle d, \pi^\ell, k \rangle$. There will be exactly one such tuple for each unique label ℓ, and each label will represent a single state in the automaton. We also

Derivative $\mathcal{D} : \mathcal{T}_\ell^* \to \mathcal{P}(\mathcal{T}_\ell^* \times \mathcal{T}_{\pi^\ell} \times \mathcal{T}_{\text{pred}}^*)$

$$
\begin{array}{lcl}
\mathcal{D}(0) & = & \emptyset \\
\mathcal{D}(1) & = & \emptyset \\
\mathcal{D}(\alpha) & = & \emptyset \\
\mathcal{D}(\pi^\ell) & = & \{\langle 1, \pi^\ell, 1\rangle\} \\
\mathcal{D}(p+q) & = & \mathcal{D}(p) \cup \mathcal{D}(q) \\
\mathcal{D}(p \cdot q) & = & \mathcal{D}(p) \odot q \cup \mathcal{E}(p) \odot \mathcal{D}(q) \\
\mathcal{D}(p^*) & = & \mathcal{D}(p) \odot p^*
\end{array}
$$

Acceptance condition $\mathcal{E} : \mathcal{T}_\ell^* \to \mathcal{T}_{\text{pred}}^*$

$$
\begin{array}{lcl}
\mathcal{E}(0) & = & 0 \\
\mathcal{E}(1) & = & 1 \\
\mathcal{E}(\alpha) & = & \alpha \\
\mathcal{E}(\pi^\ell) & = & 0 \\
\mathcal{E}(p+q) & = & \mathcal{E}(p) + \mathcal{E}(q) \\
\mathcal{E}(p \cdot q) & = & \mathcal{E}(p) \cdot \mathcal{E}(q) \\
\mathcal{E}(p^*) & = & 1
\end{array}
$$

$$\mathcal{D}(p) \odot q = \{\langle d, \pi^\ell, k \cdot q\rangle \mid \langle d, \pi^\ell, k\rangle \in \mathcal{D}(p)\} \qquad q \odot \mathcal{D}(p) = \{\langle q \cdot d, \pi^\ell, k\rangle \mid \langle d, \pi^\ell, k\rangle \in \mathcal{D}(p)\}$$

$$
\begin{array}{rcll}
\mathcal{A}_\pi(p) & = & (S, s_0, \epsilon, \delta) & \text{Term automaton} \\
S & = & \{0\} \cup \mathsf{labels}(p) & \text{States} \\
s_0 & = & 0 & \text{Initial state} \\
\epsilon\ \ell\ t & \Leftrightarrow & t \in [\![\mathcal{E}(k_\ell)]\!](t) & \text{Acceptance condition} \\
\delta\ \ell\ t & = & \{(\sigma', \pi'^{\ell'}) \mid \langle d, \pi'^{\ell'}, k\rangle \in \mathcal{D}(k_\ell) \wedge t \in [\![d]\!](t) \wedge t\langle \sigma', \pi'^{\ell'}\rangle \in [\![\pi'^{\ell'}]\!](t)\} & \text{Transition relation}
\end{array}
$$

Term automaton trace acceptance $\boxed{\mathsf{accepts} \subseteq \mathsf{Automaton} \times S \times \mathsf{Trace} \times (\mathsf{State} \times \mathcal{T}_\pi)^*}$

$$
\begin{array}{rcll}
\mathcal{A}_\pi(p), \ell \text{ accepts } t; \bullet & \Leftrightarrow & \epsilon\ \ell\ t & \text{Accepting state} \\
\mathcal{A}_\pi(p), \ell \text{ accepts } t; \langle \sigma, \pi^{\ell'}\rangle t' & \Leftrightarrow & (\sigma, \pi^{\ell'}) \in \delta\ \ell\ t \wedge \mathcal{A}_\pi(p), \ell' \text{ accepts } t\langle \sigma, \pi^{\ell'}\rangle; t' & \text{Taking a step}
\end{array}
$$

Fig. 12. KMT partial derivatives and automata

distinguish an initial state, 0. The acceptance function for state ℓ is given by $\mathcal{E}(k)$. To compute the transition relation, we compute $\mathcal{D}(k)$ for each such tuple, which yields another set of tuples. For each tuple $\langle d', \pi'^{\ell'}, k'\rangle \in \mathcal{D}(k)$, we add a transition from state π^ℓ to state $\pi'^{\ell'}$ labeled with the term $d' \cdot \pi'^{\ell'}$. The d part is a predicate identifying when the transition activates, while the k part is the "continuation", i.e., what else in the term can be run. Since labelings are unique, we use k_ℓ to refer to the unique continuation of π^ℓ when constructing $\mathcal{A}_\pi(p)$ for a given p. We let k_0 be the continuation of the initial action, i.e., the original term p.

For example, the term $\mathsf{inc}_x{}^* \cdot \Diamond x > 2$, is first labeled as $(\mathsf{inc}_x{}^1)^* \cdot \Diamond x > 2$. We then compute $\mathcal{D}((\mathsf{inc}_x{}^1)^* \cdot \Diamond x > 2) = \{\langle 1, \mathsf{inc}^1, (\mathsf{inc}_x^1)^* \cdot \Diamond x > 2\rangle\}$. Hence, there is a transition from state 0 to state 1 with label $(1 \cdot \mathsf{inc}_x)$. Taking the derivative of the resulting value, $(\mathsf{inc}_x^1)^* \cdot \Diamond x > 2$, results in the same tuple, so there is a single transition from state 1 to itself, also labeled with $1 \cdot \mathsf{inc}_x^1$. The acceptance function for this state is given by $\mathcal{E}((\mathsf{inc}_x^1)^* \cdot \Diamond x > 2) = \Diamond x > 2$. The resulting automaton, and its minimized form, are shown in Fig. 11 (left).

Lemma 4.1 (Derivative correct). *For all programs p where each primitive action π is augmented with a unique label ℓ,*

(1) $p \equiv \mathcal{E}(p) + \sum_{\langle d, \pi^\ell, k\rangle \in \mathcal{D}(p)} d \cdot \pi^\ell \cdot k$, *and*

(2) For all labels ℓ in p, there exist unique d and k such that $\langle d, \pi^\ell, k\rangle \in \mathcal{D}(p)$.

Proof. For (1), we go by induction on p, using Denesting in the star case. For (2), let π and ℓ be given; we go by induction on p. □

Lemma 4.2 (Term automaton correct). *$tt' \in [\![k_\ell]\!](t)$ iff $\mathcal{A}_\pi(p), \ell$ accepts $t; t'$.*

Proof. By induction on the length of t', leaving t general. □

$$
\begin{array}{rcll}
\mathcal{A}_\alpha(a) & = & (S, s_0, \epsilon, \delta) & \text{Theory automaton} \\
S & = & 2^{\mathsf{sub}(a)} & \text{States} \\
s_0(t) & = & \{b \in \mathsf{sub}(a) \mid t \in [\![b]\!](t)\} & \text{Initial state selector} \\
\mathsf{serialize}(A) & = & \Pi_{a \in A} a & \text{Serialization of predicate sets} \\
\epsilon\, A\, t & \Leftrightarrow & a \in A & \text{Acceptance condition} \\
\delta\, A\, t & = & \{(\sigma, \pi, \{c \mid \forall b \in A,\ \pi \cdot c \ \mathsf{PB}^\bullet\ b \cdot \pi\}) \mid t\langle \sigma, \pi \rangle \in [\![\pi]\!](t)\} & \text{Transition relation}
\end{array}
$$

Theory automaton trace stepping $\boxed{\text{traces} \subseteq \text{Automaton}_\alpha \times S \times \text{Trace} \times (\text{State} \times \mathcal{T}_\pi)^*}$

$$
\begin{array}{rcll}
\mathcal{A}_\alpha(a), A \text{ traces } t; \bullet & \Leftrightarrow & t \in [\![\mathsf{serialize}(A)]\!](t) & \text{Stopping} \\
\mathcal{A}_\alpha(a), A \text{ traces } t; \langle \sigma, \pi \rangle t' & \Leftrightarrow & (\sigma, \pi, A') \in \delta\, A\, t \wedge \mathcal{A}_\alpha(a), A' \text{ traces } t\langle \sigma, \pi \rangle; t' & \text{Taking a step}
\end{array}
$$

Fig. 13. Theory automata

The term automaton $\mathcal{A}_\pi(p)$ is equivalent to the original policy p, but we are not yet done. The term automaton makes use of arbitrary predicates in its transitions δ and its acceptance condition ϵ. For some client theories, predicates are immediately decidable, but predicates from a theory like LTL_f (Sec. 3.4) look at more than the last state of the trace. Depending on what the automata will be used for, these complex predicates may or may not be a problem. For our use here—deciding equivalence—we must simplify complex predicates: we define separate automata for tracking which predicates hold when (Sec. 4.1.2) and then construct a quasi-intersection automaton that implements predicates in the term automaton with theory automata.

4.1.2 Theory automata. Once we've constructed the term automaton, we construct theory automata for each predicate appearing anywhere in the term automaton, whether in an acceptance or a transition condition. The theory automaton for a predicate a, written $\mathcal{A}_\alpha(a)$, tracks whether a holds so far in a trace, given some initial trace and a sequence of primitive actions. Formally, $\mathcal{A}_\alpha(a)$ is a 4-tuple $(S, s_0, \epsilon, \delta)$ where S is a set of states, s_0 is an initial state selection function, ϵ is an acceptance condition, and δ is a transition relation. The states of the theory automaton are sets of subterms of the original predicate a; when the automaton is in state $A \subseteq \mathsf{sub}(a)$, then we expect every predicate $b \in A$ to hold. The runs of the theory automaton are characterized by the traces predicate. We say traces rather than accepts because we use the theory automaton to determine which predicates hold rather than to accept or reject a trace. (The KMT automaton will use the acceptance condition ϵ.) The initial state selector starts the theory automaton's run in the state identified by those subterms satistifed by the trace so far. The term automaton will use the theory automaton to implement its complex predicates by running each theory automaton in parallel: to determine whether to take an a transition, we consult the current state A of $\mathcal{A}_\alpha(a)$ and see whether $a \in A$, i.e., does a hold in the current state?

We use *pushback* (Sec. 2.3.2) to generate the transition relation of the theory automaton, since the pushback exactly characterizes the effect of a primitive action π on predicates a: to determine if a predicate α is true after some action a, we can instead check if b is true in the previous state when we know that $\pi \cdot a\ \mathsf{PB}^\bullet\ b \cdot \pi$.

While a KMT may include an infinite number of primitive actions (e.g., $x := n$ for $n \in \mathbb{N}$ in IncNat), any given term only has finitely many. For $\mathsf{inc}_x{}^* \cdot \Diamond x > 2$, there is only a single primitive action: inc_x. For each such action π that appears in the term and each subterm s of the test $\Diamond x > 2$, we compute the pushback of π and s.

Continuing our example (Fig. 11 (middle)), there is a transition from state 2 to state 3 for action inc_x. State 3 is labeled with $\{1, x > 0, x > 1, x > 2, \Diamond x > 2\}$ and state 2 is labeled with $\{1, x > 0, x > 1\}$. We compute $\mathsf{inc}_x \cdot \Diamond x > 2$ WP $(\Diamond x > 2 + x > 1)$. Therefore, $\Diamond x > 2$ should be

$$
\begin{array}{rcll}
\mathcal{A}_{\mathsf{KMT}}(p) & = & (S, s_0, \epsilon, \delta) & \text{KMT automaton} \\
S & = & S^{\mathcal{A}_\pi(p)} \times S^{\mathcal{A}_\alpha(a_1)} \times \cdots \times S^{\mathcal{A}_\alpha(a_n)} \text{ where } a_i \in \mathcal{A}_\pi(p) & \text{States} \\
s_0(t) & = & \lambda t.\, (s_0, s_0^{\mathcal{A}_\alpha(a_1)}(t), \dots, s_0^{\mathcal{A}_\alpha(a_n)}) & \text{Initial state selector} \\
\multicolumn{3}{r}{\epsilon\; s\; t \Leftrightarrow \epsilon^{\mathcal{A}_\alpha(a_i)}\; s.i\; t \text{ where } \epsilon^{\mathcal{A}_\pi(p)}\; s\; t = a_i} & \text{Acceptance condition} \\
\delta\; s\; t & = & \{(\sigma, \pi'^{\ell'}, (\ell', \delta^{\mathcal{A}_\alpha(a_1)}\; s.1\; t, \dots, \delta^{\mathcal{A}_\alpha(a_n)}\; s.n\; t)) \mid & \\
 & & \langle a_i, \pi'^{\ell'}, k\rangle \in \mathcal{D}(k_\ell) \wedge \epsilon^{\mathcal{A}_\alpha(a_i)}\; s.i\; t \wedge t\langle\sigma', \pi'^{\ell'}\rangle \in [\![\pi'^{\ell'}]\!](t)\} & \text{Transition relation}
\end{array}
$$

KMT automaton acceptance $\boxed{\text{accepts} \subseteq \text{Automaton}_{\mathsf{KMT}} \times S \times \text{Trace} \times (\text{State} \times \mathcal{T}_\pi)^*}$

$$
\begin{array}{rcll}
\mathcal{A}_{\mathsf{KMT}}(p), s \text{ traces } t; \bullet & \Leftrightarrow & \epsilon\; s\; t & \text{Accepting state} \\
\mathcal{A}_{\mathsf{KMT}}(p), s \text{ traces } t; \langle\sigma, \pi\rangle t' & \Leftrightarrow & (\sigma, \pi, s') \in \delta\; s\; t \wedge \mathcal{A}_{\mathsf{KMT}}(p), s' \text{ accepts } t\langle\sigma, \pi\rangle; t' & \\
 & & & \text{Taking a step}
\end{array}
$$

Fig. 14. Constructing KMT automata from term and theory automata

labeled in state 3 if and only if either $\Diamond x > 2$ is labeled in state 2 or $x > 1$ is labeled in state 2. Since state 2 is labeled with $x > 1$, it follows that state 3 must be labeled with $\Diamond x > 2$.

Finally, a state is accepting in the theory automaton if it is labeled with the top-level predicate for which the automaton was built. For example, state 3 is accepting (with acceptance function [1]), since it is labeled with $\Diamond x > 2$. The acceptance condition is irrelevant for how the theory automaton itself steps—we use it in combination with the term automaton.

Lemma 4.3 (Theory automaton correct). $t \in [\![\text{serialize}(A)]\!](t) \iff \mathcal{A}_\alpha(a), A \text{ traces } t; t'$

Proof. By induction on the length of t', leaving t general. □

4.2 KMT automata

We can combine the term and theory automata to create a KMT automaton, $\mathcal{A}_{\mathsf{KMT}}(p)$. The idea is to run the term and theory automata in parallel, and replacing instances of theory tests in the acceptance and transition functions of the term automaton with the test on the current state in the theory automata. The states of the KMT automaton are of the form $(\ell, A_1, \dots, A_n)$, where ℓ is a term automaton state and each A_i is a theory automaton state for some a occurring in the term automaton. In the product state, we refer to the underlying term automaton state with $s.0$ and each A_i as $s.i$. We use superscripts to disambiguate ϵ and δ, with the un-superscripted forms referring to the KMT automaton itself.

For example, in Fig. 11, the quasi-intersected automata (right) replaces instances of the $\Diamond x > 2$ condition in state 0 of the term automaton, with the acceptance condition from the corresponding state in the theory automaton. In state (2,0) this is true, while in states (1,0) and (0,0) this is false. For transitions with the same action π, the quasi-intersection takes the conjunction of each edge's tests. Formally, we define the KMT automaton as a 4-tuple $(S, s_0, \epsilon, \delta)$, where the states are those of $\mathcal{A}_\pi(p)$ along with those of $\mathcal{A}_\alpha(a)$ for every predicate a that occurs in $\mathcal{A}_\pi(p)$. The initial state selector s_0, acceptance condition ϵ, and transition relation δ are all defined as composites of the term and theory automata, using the appropriate theory automaton to implement the transition relation δ and acceptance condition ϵ.

The KMT automaton isn't, strictly speaking, an intersection automaton: we recapitulate the logic of the term automaton but use the theory automata where the term automaton would have consulted a complex predicate. As such, our proof follows the *logic* of Lemma 4.2, but we don't actually make use of that lemma at all.

Lemma 4.4 (KMT automaton correct).
$tt' \in [\![k_\ell]\!](t)$ *and* $t \in [\![\text{serialize}(A_i)]\!](t)$ *iff* $\mathcal{A}_{KMT}(p), (\ell, A_1, \ldots, A_n)$ accepts $t; t'$.

Proof. By induction on the length of t', leaving t general and using Lemma 4.3. □

4.3 Equivalence checking using automata

To check the equivalence of two KMT terms p and q, the implementation first converts both p and q to their respective (symbolic) automata. It then determinizes the automata to ensure that all transition predicates are disjoint (we use an algorithm based on minterms [15]). After combining the theory and term automata, we now have an automaton where the actions on transitions can be viewed as distinct characters. The implementation checks for a bisimulation between the two automata in a standard way by checking if, given any two bisimilar states, all transitions from the states lead to bisimilar states [9, 24, 43].

5 IMPLEMENTATION

We have implemented our ideas in an OCaml library; Sec. 1.3 summarizes the high-level idea and gives an example library implementation for the theory of increasing natural numbers. To use a higher-order theory such as that of product theories, one need only instantiate the appropriate modules in the library:

```
module P = Product(IncNat)(Boolean)
module A = Automata(P.K)   (* automata-theoretic decision procedure *)
module D = Decide(P)       (* normalization-based decision procedure *)
let a = P.K.parse "y<1; (a=F + a=T; inc(y)); y>0" in
let b = P.K.parse "y<1; a=T; inc(y)" in
assert (A.equivalent (A.of_term a) (A.of_term b));
assert (D.equivalent a b)
```

The module `P` instantiates `Product` over our theories of incrementing naturals and booleans; the module `A` gives us an automata theory for the KMT (`P.K`) associated with `P`, and the module `D` gives a way to normalize terms based on the completeness proof. User's of the library can access these representations to perform any number of tasks such as compilation, verification, inference, and so on. For example, checking language equivalence is then simply a matter of reading in KMT terms and calling the appropriate equivalence function. Our implementation currently supports both a decision procedure based on automata and one based on the normalization term-rewriting from the completeness proof. In practice, our implementation uses several optimizations, with the two most prominent being (1) hash-consing all KAT terms to ensure fast set operations, and (2) lazy construction and exploration of automata during equivalence checking. Domain optimizations are possible, too: our satisfiability procedure for `IncNat` makes a heuristic decision between using our incomplete custom solver or Z3 [19]—our solver is much faster on its restricted domain.

5.1 Optimizations

We've implemented smart constructors, which hash-cons and also automatically rewrite common identities (e.g., constructing $p \cdot 1$ will simply return p; constructing $(p^*)^*$ will simply return p^*). Client theories can extend the smart constructors to witness theory-specific identities. Client theories can implement custom solvers or rely on Z3 embeddings—custom solvers are typically faster. These optimizations are partly responsible for the speed of our normalization routine (when it avoids the costly Denest case).

	Decision Procedure	
Benchmark	**Automata**	**Normalization**
test-in-loop	9.305 sec	0.001 sec
count-twice	0.012 sec	0.001 sec
loop-reorder-arith	6.166 sec	0.001 sec
loop-parity-swap	0.010 sec	TO
compute-bool-formula	2.659 sec	0.001 sec
population-count	21.451 sec	0.001 sec

Fig. 15. Implementation microbenchmarks

We haven't particularly optimized our automata implementation. Two particular opportunities for optimization stand out, both of which focus on reducing the state space of the theory automata. First, most client-theory predicates only consider the most recent state, in which case we need not generate a theory automaton at all. Second, the formal presentation of theory automata generates one automaton per predicate, the states of which are subsets of subterms of that predicate—an exponential blowup. While convenient for the proof, many predicates will share subterms—so we pay the cost of blowup more than once, tracking the same subterms in more than one theory automaton. We could instead generate a *single* theory automaton, where a state is a set drawn from subterms of all of the predicates in the term automaton, which would reduce some of the state-space blowup.

6 EVALUATION

We performed a few experiments to evaluate our tool on a collection of simple microbenchmarks. Fig. 15 shows the microbenchmarks, each of which performs a simple task. For instance, the `population-count` example initializes a collection of boolean variables and then counts how many are set to true using a natural number counter. It proves that, if the number is above a certain threshold, then all booleans must have been set to true. The figure also shows the time it takes to verify the equivalence of terms for each example using both the automata- and normalization-based decision procedures. We use a timeout of 5 minutes.

Interestingly, the normalization-based decision procedure is very fast in many cases. This is likely due to a combination of hash-consing and smart constructors that rewrite complex terms into simpler ones when possible, and the fact that, unlike previous KAT-based normalization proofs (e.g., [1, 32]) our normalization proof does not require splitting predicates into all possible "complete tests." However, the normalization-based decision procedure does very poorly on examples where there is a sum nested inside of a Kleene star, i.e., $(p + q)^*$. The `loop-parity-swap` benchmark is one such example – it flips the parity of a boolean variables multiple times in a loop and verifies that the end value is always the same as the initial value. In this case the normalization-based decision procedure must repeatedly invoke the Denest rewriting rule, which greatly increases the size of the term on each invocation.

On the other hand, the automata-based decision procedure easily handles the `loop-parity-swap`, terminating in all cases. It takes significantly longer on most examples due to the high cost of constructing and using theory automata for every theory predicate in the term.

7 RELATED WORK

Kozen and Mamouras's Kleene algebra with equations [35] is perhaps the most closely related work: they also devise a framework for proving extensions of KAT sound and complete. Both their work and ours use rewriting to find normal forms and prove deductive completeness. Their rewriting systems work on mixed sequences of actions and predicates, but they can only delete these sequences wholesale or replace them with a single primitive action or predicate; our rewriting system's pushback operation only works on predicates due to the trace semantics that preserves the order of actions, but pushback isn't restricted to producing at most a single primitive predicate. Each framework can do things the other cannot. Kozen and Mamouras can accommodate equations that combine actions, like those that eliminate redundant writes in KAT+B! and NetKAT [1, 29]; we can accommodate more complex predicates and their interaction with actions, like those found in Temporal NetKAT [8] or those produced by the compositional theories (Sec. 3). It may be possible to build a hybrid framework, with ideas from both. A trace semantics occurs in previous work on KAT as well [27, 32].

Kozen studies KATs with arbitrary equations $x := e$ [33], also called Schematic KAT, where e comes from arbitrary first-order structures over a fixed signature Σ. He has a pushback-like axiom $x := e \cdot p \equiv \phi[{}^{x}/_{e}] \cdot x := e$. Arbitrary first-order structures over Σ's theory are much more expressive than anything we can handle—the pushback may or may not decrease in size, depending on Σ; KATs over such theories are generally undecidable. We, on the other hand, are able to offer pay-as-you-go results for soundness and completeness as well as an automata-theoretic implementation for decidability—but only for first-order structures that admit a non-increasing weakest precondition.

Larsen et al. [37] allow comparison of variables, but this of course leads to an incomplete theory. They are, able, however, to decide emptiness of an entire expression.

Coalgebra provides a general framework for reasoning about state-based systems [34, 46, 50], which has proven useful in the development of automata theory for KAT extensions. Although we do not explicitly develop the connection in this paper, KMT uses tools similar to those used in coalgebraic approaches, and one could perhaps adapt our theory and implementation to that setting. In principle, we ought to be able to combine ideas from the two schemes into a single, even more general framework that supports complex actions *and* predicates.

Our work is loosely related to Satisfiability Modulo Theories (SMT) [20]. The high-level motivation is the same—to create an extensible framework where custom theories can be combined [41] and used to increase the expressiveness and power [52] of the underlying technique (SAT vs. KA). Some of our KMT theories implement satisfiability checking by calling out to Z3 [19].

The pushback requirement detailed in this paper generalizes the classical notion of weakest precondition [6, 21, 47]. Automatic weakest precondition generation is generally limited in the presence of loops in while-programs. While there has been much work on loop invariant inference [25, 26, 28, 31, 42, 49], the problem remains undecidable in most cases; however, the pushback restrictions of "growth" of terms makes it possible for us to automatically lift the weakest precondition generation to loops in KAT. In fact, this is exactly what the normalization proof does when lifting tests out of the Kleene star operator. The pushback operation generalizes weakest preconditions because the various PB relations can change the program itself.

The automata representation described in Sec. 4 is based on prior work on symbolic automata [15, 43, 51]. In a departure from prior work, our automata construction must account for theories with predicates that look arbitrarily far back into a trace. The separate theory and term automata we use are based on ideas from Temporal NetKAT [8].

8 CONCLUSION

Kleene algebra modulo theories (KMT) is a new framework for extending Kleene algebra with tests with the addition of actions and predicates in a custom domain. KMT uses an operation that pushes tests back through actions to go from a decidable client theory to a domain-specific KMT. Derived KMTs are sound and complete with respect to a trace semantics; we derive automata-theoretic decision procedures for the KMT in an implementation that mirrors our formalism. The KMT framework captures common use cases and can reproduce *by simple composition* several results from the literature, some of which were challenging results in their own right, as well as several new results: we offer theories for bitvectors [29], natural numbers, unbounded sets, networks [1], and temporal logic [8].

ACKNOWLEDGMENTS

Dave Walker and Aarti Gupta provided valuable advice. Ryan Beckett was supported by NSF CNS award 1703493.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 113–126.

[2] Allegra Angus and Dexter Kozen. 2001. *Kleene Algebra with Tests and Program Schematology.* Technical Report. Cornell University, Ithaca, NY, USA.

[3] Valentin Antimirov. 1995. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science* 155 (1995), 291–319.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 29–43.

[5] Jorge A. Baier and Sheila A. McIlraith. 2006. Planning with First-order Temporally Extended Goals Using Heuristic Search. In *National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 788–795. http://dl.acm.org/citation.cfm?id=1597538.1597664

[6] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 82–87.

[7] Adam Barth and Dexter Kozen. 2002. *Equational verification of cache blocking in lu decomposition using kleene algebra with tests.* Technical Report. Cornell University.

[8] Ryan Beckett, Michael Greenberg, and David Walker. 2016. Temporal NetKAT. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 386–401.

[9] Filippo Bonchi and Damien Pous. 2013. Checking NFA Equivalence with Bisimulations Up to Congruence. *SIGPLAN Not.* 48, 1 (Jan. 2013), 457–468.

[10] Eric Hayden Campbell. 2017. *Infiniteness and Linear Temporal Logic: Soundness, Completeness, and Decidability.* Undergraduate thesis. Pomona College.

[11] Eric Hayden Campbell and Michael Greenberg. 2018. Injecting finiteness to prove completeness for finite linear temporal logic. (2018). In submission.

[12] Ernie Cohen. 1994. Hypotheses in Kleene Algebra. (1994). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.6067

[13] Ernie Cohen. 1994. Lazy Caching in Kleene Algebra. (1994). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5074

[14] Ernie Cohen. 1994. *Using Kleene algebra to reason about concurrency control.* Technical Report. Telcordia.

[15] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *SIGPLAN Not.* 49, 1 (Jan. 2014), 541–553.

[16] Anupam Das and Damien Pous. 2017. A Cut-Free Cyclic Proof System for Kleene Algebra. In *Automated Reasoning with Analytic Tableaux and Related Methods*, Renate A. Schmidt and Cláudia Nalon (Eds.). Springer International Publishing, Cham, 261–277.

[17] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. 2014. Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness.. In *AAAI*. Citeseer, 1027–1033.

[18] Giuseppe De Giacomo and Moshe Y Vardi. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. Association for Computing Machinery, 854–860.

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[20] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.

[21] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457.

[22] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 279–291. https://doi.org/10.1145/2034773.2034812

[23] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–309.

[24] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 343–355.

[25] Carlo A. Furia and Bertrand Meyer. 2009. Inferring Loop Invariants using Postconditions. *CoRR* abs/0909.0884 (2009).

[26] Carlo Alberto Furia and Bertrand Meyer. 2010. *Inferring Loop Invariants Using Postconditions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 277–300.

[27] Murdoch J. Gabbay and Vincenzo Ciancia. 2011. Freshness and Name-restriction in Sets of Traces with Names. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software (FOSSACS'11/ETAPS'11)*. Berlin, Heidelberg, 365–380.

[28] Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2014. Automating Full Functional Verification of Programs with Loops. *CoRR* abs/1407.5286 (2014). http://arxiv.org/abs/1407.5286

[29] Niels Bjørn Bugge Grathwohl, Dexter Kozen, and Konstantinos Mamouras. 2014. KAT + B!. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 44, 44:1–44:10 pages.

[30] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified network controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 483–494. https://doi.org/10.1145/2462156.2462178

[31] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. 2010. Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems (APLAS'10)*. 328–343.

[32] Dexter Kozen. 2003. *Kleene algebra with tests and the static analysis of programs*. Technical Report. Cornell University.

[33] Dexter Kozen. 2004. Some results in dynamic model theory. *Science of Computer Programming* 51, 1 (2004), 3 – 22. https://doi.org/10.1016/j.scico.2003.09.004 Mathematics of Program Construction (MPC 2002).

[34] Dexter Kozen. 2017. On the Coalgebraic Theory of Kleene Algebra with Tests. In *Rohit Parikh on Logic, Language and Society*. Springer, 279–298.

[35] Dexter Kozen and Konstantinos Mamouras. 2014. Kleene Algebra with Equations. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 280–292.

[36] Dexter Kozen and Maria-Christina Patron. 2000. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In *Proceedings of the First International Conference on Computational Logic (CL '00)*. Springer-Verlag, London, UK, UK, 568–582.

[37] Kim G Larsen, Stefan Schmid, and Bingtian Xue. 2016. WNetKAT: Programming and Verifying Weighted Software-Defined Networks. In *OPODIS*.

[38] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. 2016. Event-driven Network Programming. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 369–385.

[39] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 217–230. https://doi.org/10.1145/2103656.2103685

[40] Yoshiki Nakamura. 2015. Decision Methods for Concurrent Kleene Algebra with Tests: Based on Derivative. *RAMiCS 2015* (2015), 1.

[41] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245–257.

[42] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. New York, NY, USA, 42–56.

[43] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. New York, NY, USA, 357–368.

[44] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*. 109–114. https://doi.org/10.1145/2491185.2491187

[45] Grigore Roşu. 2016. Finite-Trace Linear Temporal Logic: Coinductive Completeness. In *International Conference on Runtime Verification*. Springer, 333–350.

[46] J. J.M.M. Rutten. 1996. *Universal Coalgebra: A Theory of Systems*. Technical Report. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands.

[47] Andrew E. Santosa. 2015. Comparing Weakest Precondition and Weakest Liberal Precondition. *CoRR* abs/1512.04013 (2015).

[48] Cole Schlesinger, Michael Greenberg, and David Walker. 2014. Concurrent NetCore: From Policies to Pipelines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 11–24.

[49] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. New York, NY, USA, 88–105.

[50] Alexandra Silva. 2010. *Kleene Coalgebra*. PhD Thesis. University of Minho, Braga, Portugal.

[51] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 328–341.

[52] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 2001. A Decision Procedure for an Extensional Theory of Arrays. In *LICS*.