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. Since NetKAT demonstrated how well various extensions of KATs apply to computer networks, interest in KATs has increased greatly. Unfortunately, extending a KAT to a particular 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 notions of state, we can automatically derive a corresponding KAT's semantics, prove its equational theory sound and complete, and generate an automata-based implementation of equivalence checking. Our framework is based on *pushback*, a way of specifying how predicates and actions interact, first used in Temporal NetKAT. We offer several case studies, including theories for bitvectors, increasing natural numbers, unbounded sets and maps, temporal logic, and network protocols. Finally, we provide an OCaml implementation that closely matches the theory: with only a few declarations, users can automatically derive an automata-theoretic decision procedure for a KAT.

1 INTRODUCTION

Kleene algebras with tests (KATs) provide a powerful framework for reasoning about regularly structured programs. Able to model abstractions of programs with while loops, KATs can handle a variety of analysis tasks [3, 7, 11–13, 35] 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 [1], followed by many variations and extensions [5, 8, 22, 36, 38, 47]. However, extending a KAT remains a challenging task, requiring experts familiar with KATs and their metatheory to 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. Our goal in this paper is to democratize KATs, offering a general framework for automatically deriving sound, complete, and decidable KATs for client theories. Our theoretical framework corresponds closely to an OCaml implementation, which derives a KAT with a decision procedure from small modules specifying theories.

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). More 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 are useful for representing propositional programs with while loops: we use $\cdot$ as sequential composition, $+$ as branching (a/k/a parallel composition), and * for iteration. For example, if α and β are predicates and π and ρ are actions, then the KAT term $\alpha \cdot \pi + \neg\alpha(\beta \cdot \rho)^* \cdot \neg\beta \cdot \pi$ 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. Translating the KAT term into a While program, we write: `if` α `then` π `else { while` β `do { ` ρ ` } ; ` π `}`. Reasoning in KAT is purely propositional, and the actions and tests are opaque. We know nothing about α, β, π, or ρ, or how they might interact. For example, π might be the assignment $\mathsf{i} := \mathsf{i} + 1$ and ρ might be the test $\mathsf{i} > 100$. Clearly these ought to be related—the action π can affect the

truth of ρ. 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, 5, 8, 29, 34].

As an example, 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 the history; 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 topmost packet to have a new destination IP address and port, and then saves the current packet state. The NetKAT paper goes on to explicitly restate the KAT equational theory along with custom equations for the new primitive forms, prove the theory's soundness, and then devise a novel normal form to reduce NetKAT to an existing KAT with a known completeness result. Later papers [21, 51] then developed the NetKAT automata theory used to compile of NetKAT programs into forwarding tables and to verify existing networks.

We aim to make it easier to define new KATs. Our theoretical framework and its corresponding implementation allow for quick and easy derivation of sound and complete KATs with automata-theoretic decision procedures when given arbitrary domain-specific theories.

How do we build our KATs? Our framework for deriving Kleene algebras with tests requires, at a minimum, custom predicates and actions along with a description of how these apply to some notion of state. We call these parts the *client theory*, and we call the client theory's predicates and actions "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.

Our framework hinges on an operation relating predicates and operations called *pushback*, first used to prove relative completeness for Temporal NetKAT [8]. Given a primitive action π and a primitive predicate α, the pushback operation tells us how to go from $\pi \cdot \alpha$ to some set of terms: $\sum_{i=0}^{n} \alpha_i \cdot \pi = \alpha_0 \cdot \pi + \alpha_1 \cdot \pi + \ldots$. 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. Pushback allows us to take an arbitrary term and normalize into a form where *all* of the predicates appear only at the front of the term, a convenient representation both for our completeness proof (Section 3.4) and our automata-theoretic implementation (Sections 5 and 6).

The quid pro quo. Our method takes a client theory $\mathcal{T}$ of custom primitive predicates and actions; from the client theory we generate a KMT, $\mathcal{T}^*$, on top of $\mathcal{T}$. Depending on what we know about $\mathcal{T}$, we can prove a variety of properties of $\mathcal{T}^*$; in dependency order:

(1) As a baseline, we need a notion of state and a way to assign meaning to primitive operations; we can then define a semantics for $\mathcal{T}^*$ that denotes each term as a function from a trace of states to a set of traces of states and a log of actions (Section 3.1).
(2) Our resulting semantics is sound as a KAT, with sound client equations added to account for primitives (Section 3.2; Theorem 3.1).
(3) If the client theory can define an appropriate pushback operation, we can define a normalization procedure for $\mathcal{T}^*$ (Section 3.3).

(4) If $\mathcal{T}$ is deductively complete, can decide satisfiability of predicates, and satisfies the pushback requirements, then the equational theory for $\mathcal{T}^*$ is complete and decidable given the trace-based interpretation of actions (Section 3.4; Theorem 3.8), and we can derive both an automata-theoretic model of $\mathcal{T}^*$, and an implementation in OCaml that can decide equivalence of terms in $\mathcal{T}^*$ (Sections 5 and 6).

What are our contributions?

- A new framework for defining KATs and proving their metatheory, with a novel, explicit development of the normalization procedure used in completeness (Section 3).
- Several case studies of this framework (Section 4), including a strengthening of Temporal NetKAT's completeness result, theories for unbounded state (naturals, sets, and maps), and distributed routing protocols.
- An automata theoretic account of our proof technique, which can inform compilation strategies for, e.g., NetKAT and enable equivalence checking (Section 5).
- An implementation of our framework (Section 6) which follows the proofs directly, automatically deriving automata-theoretic decision procedures for client theories.

Finally, our framework offers a new way in to those looking to work with KATs. Researchers familiar with inductive relations from, e.g., type theory and semantics, will find a familiar friend in our generalization of the pushback operation—we define it as an inductive relation.

2 MOTIVATION AND INTUITION

Before getting into the technical details, we offer an overview of how KATs are used (Section 2.1), what kinds of KMTs we can define (Section 2.2), and an extended networking example (Section 2.3).

2.1 Modeling While programs

Historically, KAT has been used to model the behavior of simple While programs. The Kleene star operator (p^*) captures the iterative behavior of while loops, and tests model conditionals in if statements and loop guards. For example, consider the program $\mathrm{P}_{\mathsf{nat}}$ (Figure 1a), a short loop over two natural-valued variables. 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 original loop is equivalent to its unfolding, i.e.:

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

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—to reason using this equivalence, we must add it as an equational assumption. Reasoning about the particular values of the variable $\mathtt{i}$ over time in $\mathrm{P}_{\mathsf{nat}}$ demands some number of relevant equational assumptions.

While purely abstract reasoning with KAT can often work for particular programs, it requires that we know exactly which equational assumptions we need on a per-program basis. Yet the ability to reason about the equivalence of programs in the presence of particular domains (such as the domain of natural numbers with addition and comparison) is important in order to model many real programs and domain-specific languages. Can we come up with theories that allow us to reason in a general way, and not per-program? Yes: we can build our own KAT, adding domain-specific

```
assume i < 50
while (i < 100) do
  i := i + 1
  j := j + 2
end
assert j > 100
```

(a) P_{nat}

```
assume 0 ≤ j < 4
while (i < 10) do
  i := i + 1
  j := (j << 1) + 3
  if i < 5 then
    insert(X, j)
end
assert in(X, 9)
```

(b) P_{set}

```
i := 0
parity := false
while (true) do
  odd[i] := parity
  i := i + 1
  parity := !parity
end
assert odd[99]
```

(c) P_{map}

Fig. 1. Example simple while programs.

equational rules for our actions. Such an approach is taken by NetKAT [1], which adds the "packet axioms" for reasoning about packets as they move through the network. Since NetKAT's equational theory has these packet axioms baked in, there's no need for per-program reasoning. But NetKAT's generality comes at the cost of proving its metatheory and developing an implementation—a high barrier to entry for those hoping to adapt KAT to their needs.

Our framework for Kleene algebras modulo theories (KMTs) allow us to derive metatheory and implementation for KATs based on a given theory. KMTs offer the best of both worlds: obviating both per-program reasoning and the need to deeply understand KAT metatheory and implementation.

2.2 Building new theories

We offer some cartoons of KMTs here; see Section 4 for technical details.

We can model the program P_{nat} (Figure 1a) by introducing a new client theory with actions $x := n$ and $x := x + 1$ and a new test $x > n$ for some collection of variables x and natural number constants n. For this theory we can add axioms like the following (where x ≠ y):

$$\begin{aligned}(x := n) \cdot (x > m) &\equiv (n > m) \cdot (x := n)\\ (x := x + 1) \cdot (y > n) &\equiv (y > n) \cdot (x := x + 1)\\ (x := x + 1) \cdot (x > n) &\equiv (x > n - 1) \cdot (x := x + 1)\end{aligned}$$

To derive completeness and decidability for the resulting KAT, the client must know how to take one of their primitive actions π (here, either $x := n$ or $x := x + 1$) and any primitive predicate α (here, just $x > n$) and take $\pi \cdot \alpha$ and push the test back, yielding an equivalent term $b \cdot \pi$, such that $\pi \cdot \alpha \equiv b \cdot \pi$ and b is in some sense no larger than α. The client's *pushback* operation is a critical component of our metatheory. Here, the last axiom shows how to push back a comparison test through an increment action and generate predicates that do not grow in size: $x > n-1$ is a "smaller" test than $x > n$. Given, say, a Presburger arithmetic decision procedure, we can automatically verify properties, like the assertion $j > 100$ that appears in P_{nat}, for programs with while loops.

Consider P_{set} (Figure 1b), a program defined over both naturals and a *set* data structure with two operations: insertion and membership tests. The insertion action $\texttt{insert}(x, j)$ inserts the value of an expression (j) into a given set (x); the membership test $\texttt{in}(x, c)$ determines whether a constant (c) is included in a given set (x). An axiom characterizing pushback for this theory has the form:

$$\mathsf{insert}(x, e) \cdot \mathsf{in}(x, c) \equiv ((e = c) + \mathsf{in}(x, c)) \cdot \mathsf{insert}(x, e)$$

Our theory of sets works for expressions e taken from another theory, so long as the underlying theory supports tests of the form $e = c$. For example, this would work over the theory of naturals since a test like $j = 10$ can be encoded as $(j > 9) \cdot \neg(j > 10)$.

Finally, $\mathrm{P}_{\mathsf{map}}$ (Figure 1c) uses a combination of mutable *boolean* values and a *map* data structure. Just as before, we can craft custom theories for reasoning about each of these types of state. For booleans, we can add actions of the form $b := \mathsf{true}$ and $b := \mathsf{false}$ and tests of the form $b = \mathsf{true}$ and $b = \mathsf{false}$. The axioms are then simple equivalences like $(b := \mathsf{true} \cdot b = \mathsf{false}) \equiv 0$ and $(b := \mathsf{true} \cdot b = \mathsf{true}) \equiv (b := \mathsf{true})$. To model map data structures, we add actions of the form $\mathtt{X}[e] := e$ and tests of the form $\mathtt{X}[c] = c$. Just as with the set theory, the map theory is parameterized over other theories, which can provide the type of keys and values—here, integers and booleans. In $\mathrm{P}_{\mathsf{map}}$, the odd map tracks whether certain natural numbers are odd or not by storing a boolean into the map's index. A sound axiom characterizing pushback in the theory of maps has the form:

$$(\mathtt{X}[e1] := e2 \cdot \mathtt{X}[c1] = c2) \;\equiv\; (e1 = c1 \cdot e2 = c2 + \neg(e1 = c1) \cdot \mathtt{X}[c1] = c2) \cdot \mathtt{X}[e1] := e2$$

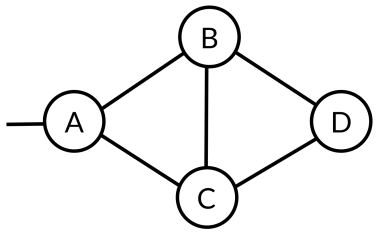
Each of the theories we have described so far—naturals, sets, booleans, and maps—have tests that only examine the *current* state of the program. However, we need not restrict ourselves in this way. Primitive tests can make dynamic decisions or assertions based on any previous state of the program. As an example, consider the theory of past-time, finite-trace linear temporal logic (LTL_f) [15, 16]. Linear temporal logic introduces new operators such as: $\bigcirc a$ (in the last state a), $\Diamond a$ (in some previous state a), and $\Box a$ (in every state a); we use finite-time LTL because finite traces are a reasonable model in most domains modeling programs. As a use of LTL_f, we may want to check for $\mathrm{P}_{\mathsf{nat}}$ (Figure 1a) that, before the last state, the variable $\mathtt{j}$ was always less than or equal to 200. We can capture this with the test $\bigcirc\Box(\mathtt{j} \leq 200)$. For LTL_f, our axioms include equalities like $p \cdot \bigcirc a \equiv a \cdot p$ and $\Box a \equiv a \cdot \bigcirc\Box a$. We can use these 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.

Finally, we can encode NetKAT, a system that extends KAT with actions of the form $f \leftarrow v$, where some value v is assigned to one of a finite number of fields f, and tests of the form $f = v$ where field f is tested for value v. It also includes a number of axioms such as $f \leftarrow v \cdot f = v \equiv f \leftarrow v$. The NetKAT axioms can be captured in our framework with minor changes. Further extending NetKAT to Temporal NetKAT is captured trivially in our framework as an application of the LTL_f theory to NetKAT's theory, deriving Beckett et al.'s [8] completeness result compositionally (in fact, we can strengthen it—see Section 4.6).



(a) Sample network with policy on C

```
A := 0
B := ∞
C := ∞
D := ∞
while (true) do
  B := min+(A, C, D)
  C := min+(A, B, D)
  D := min+(B, C)
end
```

(b) P_{SP}, default policy

```
A := (0, true)
B := (0, false)
C := (0, false)
D := (0, false)
while (true) do
  updateB
  updateC
  updateD
end
```

(c) P_{BGP}, local policies

Fig. 2. An example network and models of BGP routing.

2.3 A case study: network routing protocols

As a final example demonstrating the kinds of theories supported by KMT, we turn our attention to modeling network routing protocols. While NetKAT uses Kleene algebra to define simple, stateless forwarding tables of networks, the most common network routing protocols are distributed algorithms that actually compute paths in a network by passing messages between devices. As an example the Border Gateway Protocol (BGP) [44], which allows users to define rich routing policy, has become the de facto internet routing protocol used to transport data between between autonomous networks under the control of different entities (e.g. Verizon, Comcast). However, the combination of the distributed nature of BGP, the difficulty of writing policy per-device, and the fact that network devices can and often do fail [27] all contribute to the fact that network outages caused by BGP misconfiguration are common [2, 28, 30, 37, 40, 49, 53]. By encoding BGP policies in our framework, it immediately follows that we can decide properties about networks running BGP such as “will router A always be able to reach router B after at most 1 link failure".

Figure 2a shows an example network that is configured to run BGP. In BGP, devices exchange messages between neighbors to determine routes to a destination. In the figure, router A is connected to an end host (the line going to the left) and wants to tell other routers how to get to this destination.

In the default behavior of the BGP protocol, each router selects the shortest path among all of its neighbors and then informs each of its neighbors about this route (with the path length increased by one). In effect, the devices will compute the shortest paths through the network in a distributed fashion. We can model shortest paths routing in a KMT using the theory of natural numbers: in P_{SP} (Figure 2b), each router maintains a distance to the destination. Since A knows about the destination, it will start with a distance of 0, while all other routers start with distance ∞. Then, iteratively, each other router updates its distance to be 1 more than the minimum of each of its peers, which is captured by the `min+` operator. The behavior of `min+` can be described by pushback equivalences like:

$$B := \texttt{min+}(A, C, D) \cdot B < 3 \;\equiv\; (A < 2 + C < 2 + D < 2) \cdot B := \texttt{min+}(A, C, D)$$

BGP gets interesting when users go beyond shortest path routing and also define router-local policy. In our example network, router C is configured with local policy (Figure 2a): router C will block messages received from D and will prioritize paths received from neighbor B over those from A (using distance as a tie breaker). In order to accommodate this richer routing behavior, we must extend our model to P_{BGP} (Figure 2c). Now, each router is associated with a variable storing a tuple of both the distance and whether or not the router has a path to the destination; we write C_1 for

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

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

Actions

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

Fig. 3. Generalized KAT syntax (client parts highlighted)

the "does C have a path" boolean and C_0 for the length of that path, if it exists. We can then create a separate update action for each device in the network to reflect the semantics of the device's local policy (updateC, etc.). Further, suppose we have a boolean variable $\mathsf{fail}_{X,Y}$ for each link between routers X and Y indicating whether or not the link is failed. The update action for router C's local policy can be captured with the following type of equivalence:

$$\mathsf{updateC} \cdot C_0 < 3 \;\equiv\; (\neg\mathsf{fail}_{A,C} \cdot (\neg B_1 + \mathsf{fail}_{B,C}) \cdot A_1 \cdot (A_0 < 2) + \neg\mathsf{fail}_{B,C} \cdot B_1 \cdot (B_0 < 2)) \cdot \mathsf{updateC}$$

In order for router C to have a path length < 3 to the destination after applying the local update function, it must have either been the case that B did not have a route to the destination (or the B-C link is down) and A had a route with length < 2 and the A-C link is not down, or B had a route with length < 2 and the B-C link is not down. Similarly, we would need an axiom to capture when router C will have a path to the destination based on equivalences like: $\mathsf{updateC} \cdot C_1 \equiv (A_1 \cdot \neg\mathsf{fail}_{A,C} + B_1 \cdot \neg\mathsf{fail}_{B,C}) \cdot \mathsf{updateC}$—C has a path to the destination if any of its neighbors has a path to the destination and the corresponding link is not failed.

It is now possible to ask questions such as "if there is any single network link failure, will C ever have a path with length greater than 2?". Assuming the network program is encoded as ρ, we can answer this question by checking language non-emptiness for

$$(\mathsf{fail}_{A,C} \cdot \neg\mathsf{fail}_{B,C} + \neg\mathsf{fail}_{A,C} \cdot \mathsf{fail}_{B,C}) \cdot \rho \cdot (C_0 > 2) \;\equiv\; 0$$

While we have in a sense come back to a per-program world—P_{BGP} requires definitions and axioms for each router's local policy—we can reason in a *very* complex domain.

3 LIFTING PUSHBACK TO SOUNDNESS AND COMPLETENESS

This section details the structure of our framework for defining a KAT in terms of a client theory. While we have striven to make this section accessible to readers without much familiarity with KATs, those completely new to KATs may do well to skip directly to our case studies (Section 4). Throughout this paper, we mark those parts which must be provided by the client by highlighting them in marmalade.

We derive a KAT $\mathcal{T}^*$ (Figure 3) on top of a client theory $\mathcal{T}$ where $\mathcal{T}$ has two fundamental parts—predicates $\alpha \in \mathcal{T}_\alpha$ and actions $\pi \in \mathcal{T}_\pi$. We refer to the Boolean algebra over the client theory as $\mathcal{T}^*_{\text{pred}} \subseteq \mathcal{T}^*$.

We can provide results for $\mathcal{T}^*$ in a pay-as-you-go fashion: given a model for the predicates and actions of $\mathcal{T}$, we derive a trace semantics for $\mathcal{T}^*$ (Section 3.1); if $\mathcal{T}$ has a sound equational theory with respect to our semantics, so does $\mathcal{T}^*$ (Section 3.2); if $\mathcal{T}$ has a complete equational theory with respect to our semantics, so does $\mathcal{T}^*$ (Section 3.4); and finally, with just a few lines of code about the structure of $\mathcal{T}$, we can provide an automata-theoretic implementation of $\mathcal{T}^*$ (Section 5).

The key to our general, parameterized proof is a novel *pushback* operation (Section 3.3.2): given an understanding of how to push primitive predicates back to the front of a term, we can derive a method for finding normal forms for our completeness proof (Section 3.4).

3.1 Semantics

The first step in turning the client theory $\mathcal{T}$ into a KAT is to define a semantics (Figure 4). 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) an 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, returning true or false. Predicates can examine the entire trace—no such predicates exist in NetKAT, but those in Temporal NetKAT do. 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 semantics, though we have skipped the formal detail here. Such a feature is particularly useful for defining primitive predicates that draw from, e.g., temporal logic (see Section 4.8).

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.

3.2 Soundness

Proving soundness is relatively straightforward: we only depend on the client's act and pred functions, and none of our KAT axioms (Figure 4) even mention the client's primitives. We do need to use several KAT theorems in our completeness proof (Figure 4, Consequences), the most complex being star expansion (Star-Expand), which we take from Temporal NetKAT [8]. We believe the pushback negation theorem (Pushback-Neg) is novel. Our soundness proof naturally enough requires that any equations the client theory adds need to be sound in our trace semantics.

Theorem 3.1 (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] □

3.3 Towards completeness: normalization via pushback

In order to prove completeness (Section 3.4), we reduce our KAT terms to a more manageable subset, which we call normal forms. Normalization happens via a pushback operation to that translates 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 (Definition 3.4). 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 algebras.

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 (Section 3.3.1); and

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

Trace definitions

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

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

Trace semantics

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

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

Kleene Algebra

$p + (q + r) \equiv (p + q) + r$	KA-Plus-Assoc
$p + q \equiv q + p$	KA-Plus-Comm
$p + 0 \equiv p$	KA-Plus-Zero
$p + p \equiv p$	KA-Plus-Idem
$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$	KA-Seq-Assoc
$1 \cdot p \equiv p$	KA-Seq-One
$p \cdot 1 \equiv p$	KA-One-Seq
$p \cdot (q + r) \equiv p \cdot q + p \cdot r$	KA-Dist-L
$(p + q) \cdot r \equiv p \cdot r + q \cdot r$	KA-Dist-R
$0 \cdot p \equiv 0$	KA-Zero-Seq
$p \cdot 0 \equiv 0$	KA-Seq-Zero
$1 + p \cdot p^* \equiv p*$	KA-Unroll-L
$1 + p^* \cdot p \equiv p*$	KA-Unroll-R
$q + p \cdot r \le r \ \to \ p^* \cdot q \le r$	KA-LFP-L
$p + q \cdot r \le q \ \to \ p \cdot r^* \le q$	KA-LFP-R

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

Boolean Algebra

$a + (b \cdot c) \equiv (a + b) \cdot (a + c)$	BA-Plus-Dist
$a + 1 \equiv 1$	BA-Plus-One
$a + \neg a \equiv 1$	BA-Excl-Mid
$a \cdot b \equiv b \cdot a$	BA-Seq-Comm
$a \cdot \neg a \equiv 0$	BA-Contra
$a \cdot a \equiv a$	BA-Seq-Idem

Consequences

$p \cdot a \equiv b \cdot p \ \to$	
$\quad p \cdot \neg a \equiv \neg b \cdot p$	Pushback-Neg
$p \cdot (q \cdot p)^* \equiv (p \cdot q)^* \cdot p$	Sliding
$(p + q)^* \equiv q^* \cdot (p \cdot q^*)^*$	Denesting
$p \cdot a \equiv a \cdot q + r \ \to$	
$\quad p^* \cdot a \equiv (a + p^* \cdot r) \cdot q^*$	Star-Inv
$p \cdot a \equiv a \cdot q + r \ \to$	
$\quad p \cdot a \cdot (p \cdot a)^* \equiv$	
$\quad (a \cdot q + r) \cdot (q + r)^*$	Star-Expand

Fig. 4. Semantics and equational theory for $\mathcal{T}^*$

(2) a way to push predicates back through actions, to define the normalization procedure itself (Section 3.3.2).

Once we've defined our normalization procedure, we can use it prove completeness (Section 3.4).

3.3.1 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. For example, we can normalize the term $\text{inc}_x{}^* \cdot \Diamond x > 1$ (from LTL_f over naturals) to:

$$(\Diamond x > 1 + x > 1) \cdot \text{inc}_x{}^* \ + \ x > 0 \cdot \text{inc}_x \cdot \text{inc}_x{}^* \ + \ 1 \cdot \text{inc}_x \cdot \text{inc}_x \cdot \text{inc}_x{}^*$$

Sequenced tests and test of normal forms $\boxed{\mathsf{seqs} : \mathcal{T}^*_{\mathsf{pred}} \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$

$\boxed{\mathsf{seqs} : \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}}) \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$ $\boxed{\mathsf{tests} : \mathcal{T}^*_{\mathsf{nf}} \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$

$$\begin{array}{rcl} \mathsf{seqs}(a \cdot b) & = & \mathsf{seqs}(a) \cup \mathsf{seqs}(b) \\ \mathsf{seqs}(a) & = & \{a\} \end{array} \qquad \begin{array}{rcl} \mathsf{seqs}(A) & = & \bigcup_{a \in A} \mathsf{seqs}(a) \\ \mathsf{tests}(\sum a_i \cdot m_i) & = & \{1\} \cup \{a_i\} \end{array}$$

Subterms $\boxed{\mathsf{sub} : \mathcal{T}^*_{\mathsf{pred}} \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$ $\boxed{\mathsf{sub}_{\mathcal{T}} : \mathcal{T}_\alpha \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$ $\boxed{\mathsf{sub} : \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}}) \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$

$$\begin{array}{rcl} \mathsf{sub}(0) & = & \{0\} \\ \mathsf{sub}(1) & = & \{0, 1\} \\ \mathsf{sub}(\alpha) & = & \{0, 1, \alpha\} \cup \mathsf{sub}_{\mathcal{T}}(\alpha) \end{array} \qquad \begin{array}{rcl} \mathsf{sub}(\neg a) & = & \{0, 1\} \cup \mathsf{sub}(a) \cup \{\neg b \mid b \in \mathsf{sub}(a)\} \\ \mathsf{sub}(a + b) & = & \{a + b\} \cup \mathsf{sub}(a) \cup \mathsf{sub}(b) \\ \mathsf{sub}(a \cdot b) & = & \{a \cdot b\} \cup \mathsf{sub}(a) \cup \mathsf{sub}(b) \end{array}$$

$$\mathsf{sub}(A) = \bigcup_{a \in A} \mathsf{sub}(a)$$

Maximal tests $\boxed{\mathsf{mt} : \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}}) \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$ $\boxed{\mathsf{mt} : \mathcal{T}^*_{\mathsf{nf}} \to \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})}$

$$\mathsf{mt}(A) = \{b \in \mathsf{seqs}(A) \mid \forall c \in \mathsf{seqs}(A),\ c \neq b \Rightarrow b \notin \mathsf{sub}(c)\} \qquad \mathsf{mt}(x) = \mathsf{mt}(\mathsf{tests}(x))$$

Maximal subterm ordering $\boxed{\preceq, \prec, \approx \subseteq \mathcal{T}^*_{\mathsf{nf}} \times \mathcal{T}^*_{\mathsf{nf}}}$

$$x \preceq y \iff \mathsf{sub}(\mathsf{mt}(x)) \subseteq \mathsf{sub}(\mathsf{mt}(y)) \qquad x \prec y \iff \mathsf{sub}(\mathsf{mt}(x)) \subsetneq \mathsf{sub}(\mathsf{mt}(y))$$

$$x \approx y \iff x \preceq y \wedge y \preceq x$$

Fig. 5. Maximal tests and the maximal subterm ordering

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!

To prove that our normalization algorithm is correct, we define the *maximal subterm ordering*, which serves as our termination measure. 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 $\mathcal{T}_{\mathsf{NF}}$.

The definitions for the maximal subterm ordering are complex (Figure 5), but the intuition is: seqs gets all the tests out of a predicate; tests gets all the predicates out of a normal form; sub gets subterms; mt gets "maximal" tests that cover a whole set of tests; we lift mt to work on normal forms by extracting all possible tests; the relation $x \preceq y$ means that y's maximal tests include all of x's maximal tests. Maximal tests indicate which test to push back next in order to make progress

towards normalization. For example, the subterms of $\Diamond x > 1$ are $\{\Diamond x > 1, x > 1, x > 0, 1, 0\}$, which represents the possible tests that might be generated pushing back $\Diamond x > 1$; the maximal tests of $\Diamond x > 1$ are just $\{\Diamond x > 1\}$; the maximal tests of the set $\{\Diamond x > 1, x > 0, y > 6\}$ are $\{\Diamond x > 1, y > 6\}$ since these tests are not subterms of any other test. Therefore, we can choose to push back either of $\Diamond x > 1$ or $y > 6$ next and know that we will continue making progress towards normalization.

Definition 3.2 (Well behaved subterms). The function $\text{sub}_{\mathcal{T}}$ is *well behaved* when it uses sub in a structurally decreasing way and for all $a \in \mathcal{T}^*_{\text{pred}}$ when (1) if $b \in \text{sub}_{\mathcal{T}}(a)$ then $\text{sub}(b) \subseteq \text{sub}_{\mathcal{T}}(a)$ and (2) if $b \in \text{sub}_{\mathcal{T}}(a)$, then either $b \in \{0, 1, a\}$ or b precedes a in a global well ordering of predicates.

In most cases, it's sufficient to use the size of terms as the well ordering, but as we develop higher-order theories, we use a lexicographic ordering of "universe level" paired with term size. Throughout the following, we assume that $\text{sub}_{\mathcal{T}}$ is well behaved.

We can take a normal form x and *split* it around a maximal test $a \in \text{mt}(x)$ such that we have a pair of normal forms: $a \cdot y + z$, where both y and z are smaller than x in our ordering, because a (1) appears at the front of y and (2) doesn't appear in z at all.

Lemma 3.3 (Splitting). *If $a \in \text{mt}(x)$, then there exist y and z such that $x \equiv a \cdot y + z$ and $y \prec x$ and $z \prec x$.*

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.

3.3.2 Pushback. In order to define normalization—necessary for completeness (Section 3.4)—the client theory must have a *pushback* operation.

Definition 3.4 (Pushback). Let the sets $\Pi_{\mathcal{T}} = \{\pi \in \mathcal{T}^*\}$ and $A_{\mathcal{T}} = \{\alpha \in \mathcal{T}^*\}$. Then the *pushback* operation of the client theory is a relation $\text{PB} \subseteq \Pi_{\mathcal{T}} \times A_{\mathcal{T}} \times \mathcal{P}(\mathcal{T}^*_{\text{pred}})$. We write the relation as $\pi \cdot \alpha \ \text{PB} \ \sum a_i \cdot \pi$ and read it as "α pushes back through π to yield $\sum a_i \cdot \pi$". We require that if $\pi \cdot \alpha \ \text{PB} \ \{a_1, \ldots, a_k\}$, then $\pi \cdot \alpha \equiv \sum_{i=1}^{k} a_i \cdot \pi$, and $a_i \preceq \alpha$.

Given the client theory's PB relation, we define a normalization procedure for $\mathcal{T}^*$ (Figure 6) by extending the client's PB relation (Figure 7). The PB relation need not be a function, nor do the the a_i need to be obviously related to α or π in any way.

The top-level normalization routine is defined by the $p \ \text{norm} \ x$ relation (Figure 6), 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.

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. We use the PB^{J} relation (Figure 6), which joins two normal forms along the lines described here; we write $x \cdot y \ \text{PB}^{\text{J}} \ z$ to mean that the concatenation of x and y is equivalent to the normal form z—the $\cdot$ is merely suggestive notation, as are other operators that appear on the left-hand side of the judgment schemata.

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 to the front. We do so with the PB^* relation (Figure 6), writing $x^* \ \text{PB}^* \ y$ to mean that the Kleene star of x is equivalent to the normal form y—the * is again

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

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

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

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

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

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

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

$$\frac{x \not\prec a \qquad x \cdot a \ \mathsf{PB}^{\mathsf{T}}\ a \cdot t + u \qquad (t + u)^* \ \mathsf{PB}^*\ y \qquad y \cdot x \ \mathsf{PB}^{\mathsf{J}}\ z}{(a \cdot x)^* \ \mathsf{PB}^*\ 1 + a \cdot z}\ \textsc{Expand}$$

$$\frac{a \notin \mathsf{mt}(z) \qquad y \not\equiv 0 \qquad y^* \ \mathsf{PB}^*\ y' \qquad x \cdot y' \ \mathsf{PB}^{\mathsf{J}}\ x' \qquad (a \cdot x')^* \ \mathsf{PB}^*\ z \qquad y' \cdot z \ \mathsf{PB}^{\mathsf{J}}\ z'}{(a \cdot x + y)^* \ \mathsf{PB}^*\ z'}\ \textsc{Denest}$$

Fig. 6. Normalization $\mathcal{T}^*$

just suggestive notation. The PB^* relation is more subtle than PB^{J}. There are four possible ways to treat x, based on how it factors via splitting (Lemma 3.3): 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 otherwise (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 PB^{J} and PB^* relations rely on others to do their work (Figure 7): the bulk of the work happens in the $\mathsf{PB}^{\bullet}$ relation, which pushes a test back through a restricted action; PB^{R} and PB^{T} are wrappers around $\mathsf{PB}^{\bullet}$ it for pushing tests back through normal forms and for pushing normal forms back through restricted actions, respectively. We write $m \cdot a \ \mathsf{PB}^{\bullet}\ y$ to mean that pushing the test a back through restricted action m yields the equivalent normal form y. The $\mathsf{PB}^{\bullet}$ relation works by analyzing both the action and the test. The client theory's PB relation is used in $\mathsf{PB}^{\bullet}$ when we try to push a primitive predicate α through a primitive action π (Prim); all other KAT predicates can be handled by rules matching on the action or predicate structure, deferring to other PB relations. To handle negation, we define a function nnf that takes a predicate and translates it to *negation normal form*, where negations only appear on primitive predicates (Figure 7); the Pushback-Neg theorem justifies this case; we use nnf to guarantee that we obey the maximal subterm ordering.

To elucidate the way $\mathsf{PB}^{\bullet}$ handles structure, suppose we have the term $(\pi_1 + \pi_2) \cdot (\alpha_1 + \alpha_2)$. One of two rules could apply: we could split up the tests and push them through individually (SeqParTest), or we could split up the actions and push the tests through together (SeqParAction). It doesn't particularly matter which we do first: the next step will almost certainly be the other rule, and in any case the results will be equivalent from the perspective of our equational theory. It *could* be the case that choosing a one rule over another could give us a smaller term, which might yield a more efficient normalization procedure. Similarly, a given normal form may have more than one

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

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

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

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

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

$$\frac{m \cdot a \ \mathrm{PB}^\bullet \ x \qquad x \prec a \qquad m^* \cdot x \ \mathrm{PB}^\mathsf{R} \ y}{m^* \cdot a \ \mathrm{PB}^\bullet \ a + y}\ \textsc{SeqStarSmaller} \qquad \frac{m \cdot a \ \mathrm{PB}^\bullet \ a \cdot t + u \qquad m^* \cdot t \ \mathrm{PB}^\mathsf{R} \ x \qquad u^* \ \mathrm{PB}^* \ y \qquad x \cdot y \ \mathrm{PB}^\mathsf{J} \ z}{m^* \cdot a \ \mathrm{PB}^\bullet \ a \cdot y + z}\ \textsc{SeqStarInv}$$

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

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

$$\begin{aligned} \mathsf{nnf}(0) &= 0 & \mathsf{nnf}(\neg 0) &= 1 \\ \mathsf{nnf}(1) &= 1 & \mathsf{nnf}(\neg 1) &= 0 \\ \mathsf{nnf}(\alpha) &= \alpha & \mathsf{nnf}(\neg\alpha) &= \neg\alpha \\ \mathsf{nnf}(a + b) &= \mathsf{nnf}(a) + \mathsf{nnf}(b) & \mathsf{nnf}(\neg\neg a) &= \mathsf{nnf}(a) \\ \mathsf{nnf}(a \cdot b) &= \mathsf{nnf}(a) \cdot \mathsf{nnf}(b) & \mathsf{nnf}(\neg(a + b)) &= \mathsf{nnf}(\neg a) \cdot \mathsf{nnf}(\neg b) \\ & & \mathsf{nnf}(\neg(a \cdot b)) &= \mathsf{nnf}(\neg a) + \mathsf{nnf}(\neg b) \end{aligned}$$

Fig. 7. Pushback for $\mathcal{T}^*$

maximal test—and therefore be splittable in more than one way (Lemma 3.3)—and it may be that different splits produce more or less efficient terms. We haven't yet studied differing strategies for pushback, but see Sections 5 and 6 for discussion of our automata-theoretic implementation.

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 3.5). Once we've established that our pushback 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 3.6) .

Theorem 3.5 (Pushback soundness). *For each of the* PB *relations, the left side is equivalent to the right side, e.g., if* $x^* \ \mathrm{PB}^* \ y$ *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 (Pushback-Neg).

□

Theorem 3.6 (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 $y \preceq x$ 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 3.3) 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 3.6 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 3.7 (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 3.6 and 3.5 in the Seq and Star case. □

The PB relations and these two separate proofs are one of the contributions of this paper: we believe it is the first time that a KAT normalization procedure has been made explicit, rather than hiding inside of completeness proofs. Temporal NetKAT, which introduced the idea of pushback, proved a more limited variation of Theorems 3.5 and 3.6 as a single theorem.

3.4 Completeness

We prove completeness—that if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$—by normalizing p and q and comparing the resulting terms, just like other KATs do [1, 8]. Our completeness proof uses the completeness of Kleene algebra (KA) as its foundation: the set of possible traces of actions performed for a restricted action in our denotational semantics is a regular language, and so the KA axioms are sound and complete for it.

Theorem 3.8 (Completeness). *If there is a complete emptiness-checking procedure for $\mathcal{T}$ predicates, 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 3.7); by soundness (Theorem 3.1), 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 $p \equiv q$.

In order to support a *syntactic* comparison between the cases of our two normal forms, we construct equivalent normal forms that perform full case analysis on each component test from both x and y. As a first step, we expand x and y into $\hat{x}$ and $\hat{y}$ of the form:

$$\begin{aligned}
\hat{x} = \quad & a_1 \cdot a_2 \cdot \cdots \cdot a_n \cdot m_1 \cdot m_2 \cdot \cdots \cdot m_n \\
+ \quad & \neg a_1 \cdot a_2 \cdot \cdots \cdot a_n \cdot m_2 \cdot \cdots \cdot m_n \\
+ \quad & a_1 \cdot \neg a_2 \cdot \cdots \cdot a_n \cdot m_1 \cdot \cdots \cdot m_n \\
+ \quad & \ldots \\
+ \quad & \neg a_1 \cdot \neg a_2 \cdot \cdots \cdot a_n \cdot m_n
\end{aligned}$$

As a second step, we add each possible expansion from y into x and vice versa. Since the client theory can decide emptiness, we can eliminate those cases where we've combined contradictory terms. It now remains to be seen that syntactically equal predicates at the front are followed

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

Pushback	**Axioms**	
$b := \text{true} \cdot b = \text{true}$ PB 1	$(b := \text{true}) \cdot (b = \text{true}) \equiv (b := \text{true})$	Set-Test-True-True
$b := \text{false} \cdot b = \text{true}$ PB 0	$(b := \text{false}) \cdot (b = \text{true}) \equiv 0$	Set-Test-False-True

Fig. 8. BitVec, bit vectors

by equivalent commands. We can do so by appealing to completeness of KA, since the traces of commands run in our semantics can be interpreted as a regular language . □

4 CASE STUDIES: CLIENT THEORIES AND HIGHER-ORDER THEORIES

We define each of the client theories and higher-order theories that were discussed in Section 2.

4.1 Bit vectors

The simplest KMT we can add is one of bit vectors: we add some finite number of bits which can be set to true or false and tested for their current value (Figure 8). 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 aren't limited to just numbers, of course; once we have bits, we can encode any size-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. Our system can *only* work with a tracing semantics—see related work (Section 7) for a comparison with Kleene algebra with equations [34], a framework that can handle non-tracing semantics.

4.2 Increasing naturals

One particularly useful client theory is IncNat, which we pronounce *increasing naturals*: fix a set of variables, $\mathcal{V}$, which range over natural numbers, and allow for increment operations and comparison to constants (Figure 9). We desugar $x < n$ to $\neg(x > n - 1)$; we write $m > n$ to mean 1 when m is greater than n and 0 otherwise and $\max(m, n)$ similarly.

The proof obligations are not particularly difficult: if we count n as part of the size of $x > n$, then $\text{sub}(x > n)$ produces appropriately "smaller" terms. The pushback's soundness is relatively easy to see, since it corresponds precisely to equivalence rules: $\text{inc}_x \cdot \text{inc}_x \cdot x > 5 \equiv x > 3 \cdot \text{inc}_x \cdot \text{inc}_x$. The deductive completeness of the model shown here can be reduced to Presburger arithmetic, though we can of course use a much simpler solver for integer interval constraints.

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

$$
\begin{array}{rcl}
\alpha & ::= & x > n \\
\pi & ::= & \mathsf{inc}_x \mid x := n \\
\mathsf{pred}(x > n, t) & = & \mathsf{last}(t)(x) > n \\
\mathsf{sub}(x > n) & = & \{x > m \mid m \leq n\}
\end{array}
\qquad
\begin{array}{rcl}
n & \in & \mathbb{N} \\
x & \in & \mathcal{V} \\
\mathsf{State} & = & \mathcal{V} \to \mathbb{N} \\
\mathsf{act}(\mathsf{inc}_x, \sigma) & = & \sigma[x \mapsto \sigma(x) + 1] \\
\mathsf{act}(x := n, \sigma) & = & \sigma[x \mapsto n]
\end{array}
$$

Pushback

$x := n \cdot (x > m)$ PB $(n > m)$
$\mathsf{inc}_y \cdot (x > n)$ PB $(x > n)$
$\mathsf{inc}_x \cdot (x > n)$ PB $(x > n - 1)$ when $n \neq 0$
$\mathsf{inc}_x \cdot (x > 0)$ PB 1

Axioms

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

Fig. 9. IncNat, increasing naturals

$$
\begin{array}{rcl}
\alpha & ::= & \alpha_1 \mid \alpha_2 \\
\pi & ::= & \pi_1 \mid \pi_2 \\
\mathsf{sub}(\alpha_i) & = & \mathsf{sub}_i(\alpha_i)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{State} & = & \mathsf{State}_1 \times \mathsf{State}_2 \\
\mathsf{pred}(\alpha_i, t) & = & \mathsf{pred}_i(\alpha_i, t_i) \\
\mathsf{act}(\pi_i, \sigma) & = & \sigma[\sigma_i \mapsto \mathsf{act}_i(\pi_i, \sigma_i)]
\end{array}
$$

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

$\pi_1 \cdot \alpha_2$ PB α_2 $\qquad$ $\pi_2 \cdot \alpha_1$ PB α_1

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

$\pi_1 \cdot \alpha_2 \equiv \alpha_2 \cdot \pi_1$ L-R-Comm
$\pi_2 \cdot \alpha_1 \equiv \alpha_1 \cdot \pi_2$ R-L-Comm

Fig. 10. $\mathsf{Prod}(\mathcal{T}_1, \mathcal{T}_2)$, products of two disjoint theories

long as the operations are monotonically increasing and invertible, we can still define a pushback and corresponding equational rules. It is *not* possible, however, to compare two variables directly with tests like $x = y$—to do so would not satisfy the pushback requirement. It would be bad if it did: the test $x = y$ can encode context-free languages! The (inadmissible!) term $x := 0 \cdot y := 0; (x := x + 1)^* \cdot (y := y + 1)^* \cdot x = y$ describes programs with balanced increments between 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.

4.3 Disjoint products

Given two client theories, we can combine them into a disjoint product theory, $\mathsf{Prod}(\mathcal{T}_1, \mathcal{T}_2)$, where the our states our products and the predicates and actions from $\mathcal{T}_1$ can't affect $\mathcal{T}_2$ and vice versa (Figure 10). 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, $\mathsf{Prod}(\mathsf{BitVec}, \mathsf{IncNat})$ 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].

$$\alpha ::= \mathsf{in}(x,e) \mid e = c \mid \alpha_e \qquad c \in C$$
$$\pi ::= \mathsf{add}(x,c) \mid \mathsf{del}(x,c) \mid \pi_e \qquad e \in \mathcal{E}$$
$$x \in \mathcal{V}$$
$$\mathsf{pred}(\mathsf{in}(x,c),t) = \mathsf{last}(t)_2(e) \in \mathsf{last}(t)_1(x) \qquad \mathsf{State} = (\mathcal{V} \to \mathcal{P}(C)) \times (\mathcal{E} \to C)$$
$$\mathsf{pred}(\alpha_e,t) = \mathsf{pred}(\alpha_e,t_2) \qquad \mathsf{act}(\mathsf{add}(x,e),\sigma) = \sigma[\sigma_1[x \mapsto \sigma_1(x) \cup \{c\}]]$$
$$\mathsf{sub}(\mathsf{in}(x,c)) = \{\mathsf{in}(x,c)\} \cup \mathsf{sub}(\neg(e=c)) \qquad \mathsf{act}(\mathsf{del}(x,e),\sigma) = \sigma[\sigma_1[x \mapsto \sigma_1(x) \setminus \{c\}]]$$
$$\mathsf{sub}(e=c) = \mathsf{sub}(e=c) \qquad \mathsf{act}(\pi_e,\sigma) = \sigma[\sigma_2 \mapsto \mathsf{act}(\pi_e,\sigma_2)]$$
$$\mathsf{sub}(\alpha_e) = \mathsf{sub}(\alpha_e)$$

Pushback extending $\mathcal{E}$

$\mathsf{add}(y,e) \cdot \mathsf{in}(x,c)$	PB $\mathsf{in}(x,c)$
$\mathsf{add}(x,e) \cdot \mathsf{in}(x,c)$	PB $(e=c) + \mathsf{in}(x,c)$
$\mathsf{add}(x,e) \cdot \alpha_e$	PB α_e
$\mathsf{del}(y,e) \cdot \mathsf{in}(x,c)$	PB $\mathsf{in}(x,c)$
$\mathsf{del}(x,e) \cdot \mathsf{in}(x,c)$	PB $\neg(e=c) \cdot \mathsf{in}(x,c)$
$\mathsf{del}(x,e) \cdot \alpha_e$	PB α_e

Axioms extending $\mathcal{E}$

$\mathsf{add}(y,e) \cdot \mathsf{in}(x,c) \equiv \mathsf{in}(x,c) \cdot \mathsf{add}(y,e)$	Add-Comm
$\mathsf{add}(x,e) \cdot \mathsf{in}(x,c) \equiv ((e=c) + \mathsf{in}(x,c)) \cdot \mathsf{add}(x,e)$	Add-In
$\mathsf{del}(y,e) \cdot \mathsf{in}(x,c) \equiv \mathsf{in}(x,c) \cdot \mathsf{del}(y,e)$	Del-Comm
$\mathsf{del}(x,e) \cdot \mathsf{in}(x,c) \equiv \neg(e=c) \cdot \mathsf{in}(x,c) \cdot \mathsf{del}(x,e)$	Del-In
$\mathsf{add}(x,e) \cdot \alpha_e \equiv \alpha_e \cdot \mathsf{add}(x,e)$	Add-E-Comm
$\mathsf{del}(x,e) \cdot \alpha_e \equiv \alpha_e \cdot \mathsf{del}(x,e)$	Del-E-Comm

Fig. 11. Set($\mathcal{E}$), unbounded sets over arbitrary expressions/constants

$$\alpha ::= x[e] = c \mid e = c \mid \alpha_e \qquad c \in C$$
$$\pi ::= x[c] := e \mid \pi_e \qquad e \in \mathcal{E}$$
$$\mathsf{pred}(x[e]=c),t) = \mathsf{last}(t)_1(\mathsf{last}(t)_2(e)) = c \qquad x \in \mathcal{V}$$
$$\mathsf{pred}(\alpha_e,t) = \mathsf{pred}(\alpha_e,t_2) \qquad \mathsf{State} = (\mathcal{V} \to C \to C) \times (\mathcal{E} \to C)$$
$$\mathsf{sub}(x[e]=c) = \{x[e]=c\} \cup \mathsf{sub}(\neg(e=c')) \qquad \mathsf{act}(x[c] := e),\sigma) = \sigma[\sigma_1[c \mapsto \sigma_2(e)]]$$
$$\mathsf{act}(\pi_e,\sigma) = \sigma[\sigma_2 \mapsto \mathsf{act}(\pi_e,\sigma_2)]$$
$$\mathsf{sub}(e=c) = \mathsf{sub}(e=c)$$
$$\mathsf{sub}(\alpha_e) = \mathsf{sub}(\alpha_e)$$

Pushback extending $\mathcal{E}$

$(x[c] := e) \cdot \alpha_e$	PB α_e
$(y[c_1] := e_1) \cdot (x[e_2] = c_2)$	PB $x[e_2] = c_2$
$(x[c_1] := e_1) \cdot (x[e_2] = c_2)$	PB $(e_2 = c_1 \cdot e_1 = c_2) + (\neg(e_2 = c_1) \cdot x[e_2] = c_2)$

Axioms extending $\mathcal{E}$

$(x[c] := e \cdot \alpha_e) \equiv (\alpha_e \cdot x[c] := e)$	E-Comm
$(y[c_1] := e_1 \cdot x[e_2] = c_2) \equiv (x[e_2] = c_2 \cdot y[c_1] := e_1)$	Map-NEq
$(x[c_1] := e_1 \cdot x[e_2] = c_2) \equiv ((e_2 = c_1 \cdot e_1 = c_2) + \neg(e_2 = c_1) \cdot x[e_2] = c_2) \cdot x[c_1] := e_1$	Map-Eq

Fig. 12. Map($\mathcal{E}$), unbounded maps over arbitrary expressions/constants

4.4 Unbounded sets

We define a KMT for unbounded sets (Figure 11), parameterized on a theory of expressions $\mathcal{E}$. The set data type supports two operations: $\mathsf{add}(x,e)$ adds the value of expression e to set x, and $\mathsf{del}(x,e)$ removes the value of expression e from set x. It also supports a single test: $\mathsf{in}(x,c)$ checks if the constant c is contained in set x.

To instantiate the Set theory, we need a few things: expressions $\mathcal{E}$, a subset of *constants* $C \subseteq \mathcal{E}$, and predicates for testing (in)equality between expressions and constants ($e = c$ and $e \neq c$). (We can't, 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: $\sigma_1 : \mathcal{V} \to \mathcal{P}(C)$ records the current sets for each set in $\mathcal{V}$, while $\sigma_2 : \mathcal{E} \to C$ evaluates expressions in each state. Since each state has its own evaluation function, the expression language can have actions that update σ_2.

For example, we can have sets of naturals by setting $\mathcal{E} ::= n \in \mathbb{N} \mid i \in \mathcal{V}'$, where $\mathcal{V}'$ 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 $(\mathsf{inc}_i \cdot \mathsf{add}(x, i))^* \cdot (i > 100) \cdot \mathsf{in}(x, 100)$ 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 $\mathsf{in}(x, e)$ formulae must come *later* in the global well ordering than any of those generated by the client theory's $e = c$ or $e \neq c$. Sets can *almost* be defined as a disjoint product of set and expression theories, except that the set theory's pushback generates terms in the expression theory.

4.5 Unbounded maps

Maps aren't much different from sets; rather than having simple membership tests, we instead check to see whether a given key maps to a given constant (Figure 12). Our writes use constant keys and expression values, while our reads use variable keys but constant values. We could have flipped this arrangement—writing to expression keys and reading from constant ones—but we cannot allow *both* reads and writes to expression keys. Doing so would allow us to compare variables, putting us in the realm of context-free languages and foreclosing on the possibility of a complete theory. We could add other operations (at the cost of even more equational rules/pushback entries), like the ability to remove keys from maps or to test whether a key is in the map or not. Just as for $\mathsf{Set}(\mathcal{E})$, we must put all $x[e] = c$ and $x[e] \neq c$ formulae *later* in the global well ordering than any of those generated by the client theory's $e = c$ or $e \neq c$.

4.6 LTLf

The most interesting higher-order theory we present is that for past-time linear temporal logic on finite traces, LTL_f (Figure 13). Our theory of LTL_f is itself parameterized on a theory $\mathcal{T}$, which introduces its own predicates and actions—any $\mathcal{T}$ predicate can appear inside of LTL_f's predicates.

LTL_f only needs 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.

The pred definitions mostly defer to the client theory's definition of pred (which may recursively reference the LTL_f pred function), unrolling $\mathcal{S}$ as it goes (LTL-Since-Unroll). The pushback operation 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 $\mathsf{PB}^\bullet$ judgment. Adding these rules leaves our judgments monotonic, and if $\pi \cdot a \ \mathsf{PB}^\bullet \ x$, then $x = \sum a_i \pi$.

The equivalences given are borrowed from Temporal NetKAT [8] and the deductive completeness result is borrowed from Campbell's undergraduate thesis, which proves deductive completeness for an axiomatic framing and then relates those axioms to our equations [10].

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

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

Pushback extending $\mathcal{T}$

$$\pi \cdot \bigcirc a \ \mathrm{PB}\ a$$

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

$$\mathrm{State} = \mathrm{State}_{\mathcal{T}}$$
$$\mathrm{pred}(\bigcirc a, \langle \sigma, l \rangle) = \mathrm{false}$$
$$\mathrm{pred}(\bigcirc a, t\langle \sigma, l \rangle) = \mathrm{pred}(a, t)$$
$$\mathrm{pred}(a \mathbin{\mathcal{S}} b, \langle \sigma, l \rangle) = \mathrm{pred}(b, \langle \sigma, l \rangle)$$
$$\mathrm{pred}(a \mathbin{\mathcal{S}} b, t\langle \sigma, l \rangle) = \mathrm{pred}(b, t\langle \sigma, l \rangle) \vee (\mathrm{pred}(a, t\langle \sigma, l \rangle) \wedge \mathrm{pred}(a \mathbin{\mathcal{S}} b, t))$$

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
$a \mathbin{\mathcal{S}} b \equiv b + a \cdot \bigcirc(a \mathbin{\mathcal{S}} b)$	LTL-Since-Unroll
$\neg(a \mathbin{\mathcal{S}} b) \equiv (\neg b) \mathbin{\mathcal{B}} (\neg a \cdot \neg b)$	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. 13. $\mathrm{LTL}_f(\mathcal{T})$, linear temporal logic on finite traces over an arbitrary theory

$$\alpha ::= f = v$$
$$\pi ::= f \leftarrow v$$
$$\mathrm{sub}(\alpha) = \{\alpha\}$$
$$\mathrm{pred}(f = v, t) = \mathrm{last}(t).f = v$$

$$\mathsf{F} = \text{packet fields}$$
$$\mathsf{V} = \text{packet field values}$$
$$\mathrm{State} = \mathsf{F} \to \mathsf{V}$$
$$\mathrm{act}(f \leftarrow v, \sigma) = \sigma[f \mapsto v]$$

Pushback

$f \leftarrow v \cdot f = v \ \mathrm{PB}\ 1$
$f \leftarrow v \cdot f = v' \ \mathrm{PB}\ 0$ when $v \neq v'$
$f' \leftarrow v \cdot f = v \ \mathrm{PB}\ f = v$

Axioms

$f \leftarrow v \cdot f' = v' \equiv f' = v' \cdot f \leftarrow v$	PA-Mod-Comm
$f \leftarrow v \cdot f = v \equiv f \leftarrow v$	PA-Mod-Filter
$f = v \cdot f = v' \equiv 0$, if $v \neq v'$	PA-Contra
$\sum_v f = v \equiv 1$	PA-Match-All

Fig. 14. Tracing NetKAT a/k/a NetKAT without dup

4.7 Tracing NetKAT

NetKAT defines a KMT over packets, which we model as functions from packet fields to values (Figure 14). NetKAT could, in principle, be implemented as a particular instance of BitVec (Section 4.1), but we present it in full in order to simplify Temporal NetKAT (Section 4.8).

Our trace-based strategy for the semantics means we have a slightly different model from conventional NetKAT [1]. NetKAT, like KAT+B! [29], 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 dup commands to record relevant moments in time. From our 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. The upshot is that our "tracing NetKAT" has a slightly different equational theory

$$
\begin{array}{rcl}
\alpha & ::= & x < n \\
\pi & ::= & x := \mathsf{min}{+}(\vec{x}) \\
\mathsf{pred}(x < n, t) & = & \mathsf{last}(t)(x) < n \\
\mathsf{sub}(x < n) & = & \{x_i < m \mid m \leq n, x_i \in \mathcal{V}\} \\
\mathsf{act}(x := \mathsf{min}{+}(\vec{x}), \sigma) & = & \sigma[x \mapsto 1 + \mathsf{min}(\sigma(\vec{x}))]
\end{array}
\qquad
\begin{array}{rcl}
n & \in & \mathbb{N} \cup \{\infty\} \\
x & \in & \mathcal{V} \\
\mathsf{State} & = & \mathcal{V} \to \mathbb{N}
\end{array}
$$

Pushback	axioms are identical to pushback
$x := \mathsf{min}{+}(\vec{x}) \cdot (x < \infty)$	PB $\Sigma_i (x_i < \infty)$
$x := \mathsf{min}{+}(\vec{x}) \cdot (x < n)$	PB $\Sigma_i (x_i < n - 1)$

Fig. 15. SP, shortest paths in a graph

from conventional NetKAT, rejecting the following NetKAT laws as unsound for trace semantics:

$$
\begin{array}{rl}
f = v \cdot f \leftarrow v \equiv f = v & \text{PA-Filter-Mod} \\
f \leftarrow v \cdot f \leftarrow v' \equiv f \leftarrow v' & \text{PA-Mod-Mod} \\
f \leftarrow v \cdot f' \leftarrow v' \equiv f' \leftarrow v' \cdot f \leftarrow v & \text{PA-Mod-Mod-Comm}
\end{array}
$$

In principle, one can abstract our semantics' traces to find the more restricted NetKAT traces, but we can't offer any formal support in our framework for abstracted reasoning. Just as for BitVec, It is possible that ideas from Kozen and Mamouras could apply here [34]; see Section 7.

4.8 Temporal NetKAT

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

4.9 Distributed routing protocols

The theory for naturals with the min+ operator used for shortest path routing is shown in Figure 15. The theory is similar to the `IncNat` theory but for some minor differences. First, the domain is now over $\mathbb{N} \cup \{\infty\}$. Second, there is a new axiom and pushback relation relating min+ to a test of the form $x < n$. Third, the subterms function is now defined in terms of all other variables, which are infinite in principle but finite in any given term (e.g., the number of routers in a given network).

The theory for the BGP protocol instance with local router policy described in Figure 2 is now shown in Figure 16. For brevity, we only show the theory for router C in the network. The state has two parts: the first part maps each router to a pair of a natural number describing the path length to the destination for that router, and a boolean describing whether or not the router has a route to the destination; the second part maps links to a boolean representing whether the link is up or not. We require new axioms corresponding to each of the pushback operations shown. The action `updateC` commutes with unrelated tests, and otherwise behaves as described in Section 2.

5 AUTOMATA

While the deductive completeness proof from Section 3 gave a way to determine equivalence of KAT terms through a normalization rewriting, using such rewriting-based proofs as the basis of a decision procedure would be impractical. But just as pushback gave us a novel completeness proof, it can also help us develop an automata-theoretic account of equivalence.

$$
\begin{array}{rcl}
\alpha & ::= & C_0 < n \mid C_1 \mid \mathsf{fail}_{R_1,R_2} \\
\pi & ::= & \mathsf{updateC} \\
\mathsf{pred}(C_1, t) & = & \mathsf{last}(t)_1(C)_1 \\
\mathsf{pred}(C_0 < n, t) & = & \mathsf{last}(t)_1(C)_0 < n \\
\mathsf{pred}(\mathsf{fail}_{R_1,R_2}, t) & = & \mathsf{last}(t)_2(R_1, R_2) \\
\mathsf{sub}(\mathsf{fail}_{R_1,R_2}) & = & \{\mathsf{fail}_{R_1,R_2}\} \\
\mathsf{sub}(C_1) & = & \{A_1, B_1, \mathsf{fail}_{A,C}, \mathsf{fail}_{B,C}\} \\
\mathsf{sub}(C_0 < n) & = & \{A_0 < n-1, B_0 < n-1, A_1, B_1, \mathsf{fail}_{A,C}, \mathsf{fail}_{B,C}\} \\
\mathsf{act}(\mathsf{updateC}), \sigma) & = & \sigma \left[C \mapsto \begin{cases} (1+\sigma(B)_0, \mathit{true}) & \mathsf{path}(B) \\ (1+\sigma(A)_0, \mathit{true}) & \text{else if } \mathsf{path}(A) \\ (\sigma(C)_0, \sigma(C)_1) & \text{otherwise} \end{cases} \right] \\
 & & \text{where } \mathsf{path}(X) = \sigma(X)_1 \wedge \sigma((X,C))_1
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{R} & = & \text{Routers} \\
\mathsf{L} & = & \mathsf{R} \times \mathsf{R} \quad \text{Links} \\
n & \in & \mathbb{N} \\
x & \in & \mathsf{R} \\
\mathsf{State} & = & \mathsf{R} \to \mathbb{N} \times \{\mathsf{true}, \mathsf{false}\} \\
 & \times & \mathsf{L} \to \{\mathsf{true}, \mathsf{false}\}
\end{array}
$$

Pushback		axioms are identical to pushback
$(\pi \cdot \mathsf{fail}_{R_1,R_2})$	PB	fail_{R_1,R_2}
$(\mathsf{updateC} \cdot D_1)$	PB	D_1
$(\mathsf{updateC} \cdot C_1)$	PB	$\neg\mathsf{fail}_{A,C} \cdot A_1 + \mathsf{fail}_{B,C} \cdot B_1$
$(\mathsf{updateC} \cdot D_0 < n)$	PB	$(D < n)$
$(\mathsf{updateC} \cdot C_0 < n)$	PB	$\neg\mathsf{fail}_{A,C} \cdot (\neg B_1 + \mathsf{fail}_{B,C}) \cdot A_1 \cdot (A_0 < n-1) +$ $\neg\mathsf{fail}_{B,C} \cdot B_1 \cdot (B_0 < n-1)$

Fig. 16. BGP, protocol theory for router C from the network in Figure 2

Our automata theory is heavily based on previous work on Antimirov partial derivatives [4] and NetKAT's compiler [51]. We must diverge from prior work, though, to account for client theory predicates that depend on more than the last state of the trace. Our solution is adapted from the compilation strategy from Temporal NetKAT [8]: to construct an automaton for a term in a KMT, we build two separate automata—one for the policy fragment of the term and one for each predicate that occurs therein—and combine the two in a specialized intersection operation.

5.1 KMT automata, formally and by example

A *KMT automaton* is a 5-tuple $(S, s_0, \epsilon, \delta, \delta_0)$, where: the set of automata states S identifies non-initial states (unrelated to State); the *initial state* s_0 is distinguished from those in S; the *acceptance function* $\epsilon : S \to \mathcal{P}(\text{State})$ is a function identifying which theory states (in State) are accepted in each automaton state $s \in S$; the *transition function* $\delta : S \to \text{State} \to \mathcal{P}(S \times \text{Log})$ identifies successor states given an automaton and a single KMT state; and the *initial transition function* $\delta_0 : \text{Trace} \to \mathcal{P}(S \times \text{State})$ looks at a trace and identifies a set of successor states. Intuitively, the automata match traces, which are sequences of log entries: $\langle \sigma_0, \pi_1 \rangle \ldots \langle \sigma_n, \pi_n \rangle$.

Consider the KMT automaton shown in Figure 17 (right) for the term $\mathsf{inc}_x{}^* \cdot \Diamond x > 2$ taken from the $\mathrm{LTL}_f(\mathsf{IncNat})$ theory. The automaton would accept 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 automaton starts in the initial state s_0, which is (0,0) (the particular value used to represent the state, tuples in this case, is unimportant), and moves to state (1,1) for the transition labeled with $x > 0$. This predicate describes a set of theory states including the one where x is 1. The automaton then moves to state (3,1) and then (4,1) unconditionally for the inc_x action, which corresponds to actions in the log entries of the trace. The acceptance function

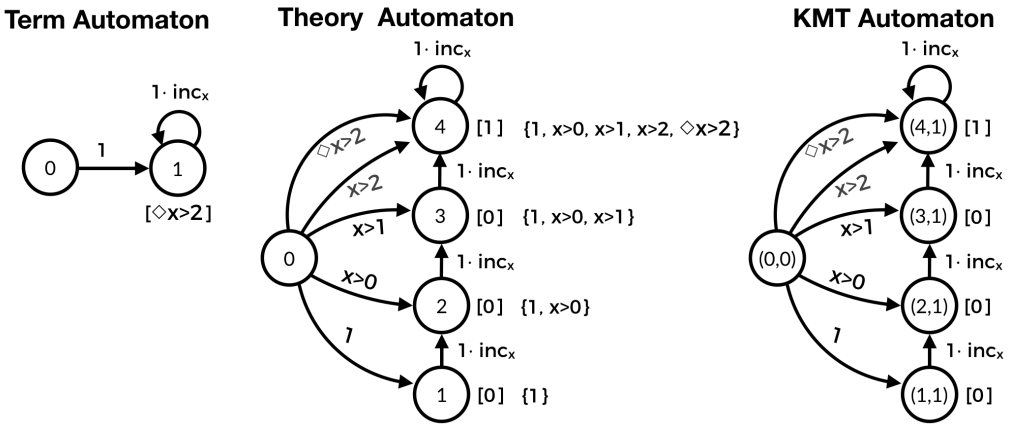


Fig. 17. Automata construction for $\texttt{inc}_x{}^* \cdot \Diamond x > 2$ in the theory of LTL_f (IncNat).

assigns state (4,1) the condition 1, meaning that all theory states are accepted; no other states are accepting, i.e., the acceptance function assigns them the condition 0.

The transition function δ takes an automaton state S and a KMT state 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 transitions as arcs between states with a pair of a KMT test and a primitive KMT action. For example, the transition from state (1,1) to (2,1) is captured by the term $1 \cdot \text{inc}_x$, which captures the effect of updating the theory domain state in the log by incrementing the value of x. Finally, the initial transition function δ_0 is similar to δ, but accounts for the fact that there may be an initial trace of log items in the initial state (as in LTL_f). For example, the transition from (0,0) to (4,1) is taken if either the initial trace has $x > 2$ in the current state or at some point in the past (represented with multiple transitions).

Taken all together, our KMT automaton captures the fact that there are 4 interesting cases for the term $\texttt{inc}_x{}^* \cdot \Diamond x > 2$. 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 (4,1) 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 (3,1). If we see at least one more increment command, then we move to state (4,1) where the trace will be accepted no matter what. If the initial trace has $x > 0$, we move to state (2,1) 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 (1,1) and must see at least 3 increment commands before accepting.

5.2 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 arbitrary predicates are placed as acceptance conditions. Such a symbolic automaton can be unwieldy—for example, the term automaton in Figure 17 (left) has a temporal predicate as an acceptance condition—challenging to reason about! We therefore find every predicate mentioned in the term automaton and construct a corresponding *theory automaton*, using pushback to move tests to the front of the automaton. We finally intersect these two to form a combined automaton with simple acceptance conditions (0 or 1).

Derivative $\mathcal{D} : \mathcal{T}_\pi \to \mathcal{P}(\mathcal{T}_\pi \times \mathbb{N} \times \mathcal{T}_\pi)$		
$\mathcal{D}(0)$	=	$\emptyset$
$\mathcal{D}(1)$	=	$\emptyset$
$\mathcal{D}(\alpha)$	=	$\emptyset$
$\mathcal{D}(\pi^\ell)$	=	$\{\langle 1, \ell, 1\rangle\}$
$\mathcal{D}(p + q)$	=	$\mathcal{D}(p) \cup \mathcal{D}(q)$
$\mathcal{D}(p \cdot q)$	=	$\mathcal{D}(p) \odot q \cup \mathcal{E}(p) \odot \mathcal{D}(q)$
$\mathcal{D}(p^*)$	=	$\mathcal{D}(p) \odot p^*$

Acceptance condition $\mathcal{E} : \mathcal{T}_\pi \to \mathcal{T}_\alpha$		
$\mathcal{E}(0)$	=	0
$\mathcal{E}(1)$	=	1
$\mathcal{E}(\alpha)$	=	α
$\mathcal{E}(\pi^\ell)$	=	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

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

Fig. 18. KMT partial derivatives

5.2.1 Term automata. Given a KMT term p, we start by annotating each occurrence of a theory action π in p with a unique label ℓ. Then we take the partial derivative of p by computing $\mathcal{D}(p)$. Figure 18 gives the definition of the derivative function. The derivative computes a set of *linear forms*—tuples of the form $\langle d, \ell, k\rangle$. There will be exactly one such tuple for each unique label ℓ, and each label will represent a single state in the automaton. The acceptance function for state ℓ is given by $\mathcal{E}(k)$. To compute the transition function, for each such tuple, we then compute $\mathcal{D}(k)$, which will return another set of tuples. For each such tuple: $\langle d', \ell', k'\rangle \in \mathcal{D}(k)$, we add a transition from state ℓ to state ℓ' 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.

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 $\mathcal{D}((\mathsf{inc}_x{}^1)^* \cdot \Diamond x > 2) = \{\langle 1, \mathsf{inc}^1, (\mathsf{inc}_x^1)^* \cdot \Diamond x > 2\rangle\}$. 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, which has its transition 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 is shown in Figure 17 (left). We add a transition from the initial state 0 to state 1 to represent the initial transition function.

5.2.2 Theory automata. Once we've constructed the term automaton, we construct theory automata for each predicate that appears in an acceptance or transition condition of the term automaton. The theory automaton for a predicate a tracks whether a holds so far in a trace, given some initial trace and a sequence of primitive actions. We use *pushback* (Section 3.3.2) 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.

For non-initial states, we maintain a labeling function $\mathcal{L}$, which identifies the set subterms of the top-level predicate a that hold in that state. For each subterm s, we add a transition $1 \cdot \pi$ from states x to y if $s \in \mathcal{L}(y)$ and $\pi \cdot s$ $\mathsf{PB}^\bullet$ $b \cdot \pi$, and b is true given the labeling $\mathcal{L}(x)$. For example, in the theory automaton show in Figure 17 (middle), there is a transition from state 3 to state 4 for action inc_x. State 4 is labeled with $\{1, x > 0, x > 1, x > 2, \Diamond x > 2\}$ and state 3 is labeled with $\{1, x > 0, x > 1\}$. We compute $\mathsf{inc}_x \cdot \Diamond x > 2$ PB $(\Diamond x > 2 + x > 1)$. Therefore, $\Diamond x > 2$ should be

labeled in state 4 if and only if either $\Diamond x > 2$ is labeled in state 3 or $x > 1$ is labeled in state 3. Since state 3 is labeled with $x > 1$, it follows that state 4 must be labeled with $\Diamond x > 2$.

We must again treat the initial state specially: there is a transition for each satisfiable combination of subterms of the top-level predicate a; the transition leads to a state labeled with all the other subterms implied by this subterm. For example, if $a = \Diamond x > 2$ and we are considering the subterm $x > 2$, then we know that if $x > 2$, then $x > 1$ and $\Diamond x > 2$ as well, so we construct an edge to a node with all three in its label. We can discover these implications by calling out to the client theory to check validity of the implication.

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 4 is accepting (with acceptance function [1]), since it is labeled with $\Diamond x > 2$.

5.3 KMT automata

We can combine the term and theory automata to create a KMT automaton. The idea is to intersect the two automata together, and replace instances of theory tests in the acceptance and transition functions of the term automaton with the acceptance condition for the given state in the theory automata. For example, in Figure 17, the combined automata (right) replaces instances of the $\Diamond x > 2$ condition in state 1 of the term automaton, with the acceptance condition from the corresponding state in the theory automaton. In state (3,1) this is true, while in states (2,1) and (1,1) this is false. For transitions with the same action π, the intersection takes the conjunction of each edge's tests.

5.4 Equivalence checking

Due to space constraints, we only briefly summarize the ideas behind equivalence checking. To check the equivalence of two KMT terms p and q, we first convert both p and q to their respective (symbolic) automata. We then determinize the automata, using an algorithm based on minterms [14], to ensure that all transition predicates are disjoint, and the typical automata powerset construction. We then check for a bisimulation between the two automata [9, 21, 43] by checking if, given any two bisimilar states, all transitions from the states lead to bisimilar states.

6 IMPLEMENTATION

We have implemented our ideas in OCaml. Users of the library write theory implementations by defining new OCaml modules that define the types of actions and tests, and functions for parsing, computing subterms, calculating pushback for primitive actions and predicates, and deciding theory satisfiability functions.

Figure 19 shows an example library implementation of the theory of naturals. The implementation starts by defining a new, recursive module called `IncNat`. Recursive modules are useful, because it allows the author of the module to access the final KAT functions and types derived after instantiating KA with their theory while still implementing the theory itself. For example, the module `K` on the second line gives us a recursive reference to the resulting KAT instantiated with the `IncNat` theory; such self-reference is key for theories like LTL_f, which must embed KAT predicates inside of temporal operators and call the KAT pushback while performing its own (Section 4.6). In the example, the user then defines two types: the type a for tests, and the type p for actions. Tests are of the form $x > n$ where variable names are represented with `string`s, and values with OCaml `int`s. The only action is to increment the variable, so we simply need to know the variable name.

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. In this example, the parser is extended to read actions of the form $inc(x)$ and tests of the form $x > c$. The other functions:

```
module rec IncNat : THEORY = struct
 module K = KAT(IncNat) (* recursive self reference to generated KMT *)
 type a = Gt of string * int (* alpha ::= x > n *)
 type p = string (* pi ::= inc x *)
 let parse name es = (* extensible parser *)
  match name, es with
  | "inc", [EId s1] → Action s1
  | ">", [EId s1; EId s2] → Test (Gt(s1, int_of_string s2))
  | _, _ → failwith "Cannot create theory object"
 let rec subterms (a : K.a) = (* for maximal term ordering/pushback *)
  match a with
  | Gt(_,0) → PSet.singleton a
  | Gt(v,i) → PSet.add a (subterms (Gt(v,i−1)))
 let push_back (x : K.p) (a : K.a) = (* pushback on theory terms *)
  match a with
  | Gt(_,0) → PSet.empty
  | Gt (y,v) when x = y → PSet.add  (K.theory a)  (Gt(y,v−1))
  | _ → PSet.singleton (K.theory a)
 let satisfiable (t : K.test) = ... (* decision procedure *)
end

module K = KAT(LTLF(IncNat)) (* build KAT from LTLF on incrementing naturals *)
module A = Automata(K) (* construct automata for our KAT *)
let main () =
 let a = K.parse "inc(x)*; since(true, x>2)" in
 let b = K.parse "since(true, x>2); inc(x)* + inc(x)*; x > 2; inc(x)*" in
 assert (A.equivalent (A.of_term a) (A.of_term b))
```

Fig. 19. Simplified example and use of an implementation of IncNat in OCaml.

subterms and push_back follow from the KMT theory directly. Finally, the user must implement a function that checks satisfiability of a theory test. To use a theory, one need only instantiate the appropriate modules: The module K instantiates LTLF over our theory of incrementing naturals; the module A gives us an automata theory for K. Checking language equivalence is then simply a matter of reading in a term, constructing an automata and checking equivalence. 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 for equivalence checking. Our satisfiability procedure for IncNat makes a heuristic decision between using our incomplete custom solver or Z3 [17]—our solver is much faster on its restricted domain.

7 RELATED WORK

Kozen and Mamouras's Kleene algebra with equations [34] 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 they can only delete these

sequences wholesale or replace them with a single primitive action or predicate; our rewriting system (pushback) only works on predicates due to the trace semantics that preserves the order of actions, but pushing a test back can yield something larger than a single primitive predicate. In the big picture, Kozen and Mamouras can accommodate equations that combine actions, like those that eliminate redundant writes in KAT+B! and NetKAT [1, 29], while we can accommodate complex predicates and their interaction with actions, like those found in Temporal NetKAT [8] or those produced by the theory combinators in Section 4. A trace semantics like the one described in this paper is used in previous work on KAT as well [25, 32].

Kozen studies KATs with arbitrary equations $x := e$ [33], where e comes from arbitrary first-order structures over a fixed signature Σ, He has a pushback-like axiom $x := e \cdot p \equiv \phi[xe] \cdot x := e$. 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. 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 for theories over particular first-order structures.

Coalgebra provides a general framework for reasoning about state-based systems [45, 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, Kleene algebra modulo theories uses tools similar to those used in coalgebraic approaches, and one could perhaps adapt our scheme to that setting.

Propositional dynamic logic (PDL) [20] gives an alternative approach to reasoning about regularly structured programs. It is a decidable logic with a complete axiomatization [42] that can embed Kleene algebra. However, like KAT, PDL is purely propositional and does not address the problem of reasoning within a particular domain of interest. Further PDL comes with a different set of tradeoffs: while KAT is PSPACE complete, PDL is known to be EXPTIME-complete. von Karger [54] shows how Kleene algebra can be extended with domain and codomain operators to give an algebraic approach to PDL, however he does not prove that the extension is complete, and still cannot reason about particular theories.

Our work is loosely related to Satisfiability Modulo Theories (SMT) [18]. The high-level motivation is the same—to create an extensible framework where custom theories can be combined [39] and used to increase the expressiveness and power [52] of the underlying technique (SAT vs. KA). However, the specifics vary greatly—while SMT is used to reason about the satisfiability of a formula, KMT is used to reason about the structure of the program and its interaction with tests.

The pushback requirement detailed in this paper is strongly related to the classical notion of weakest precondition [6, 19, 46]. However, 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 [23, 24, 26, 31, 41, 48], the problems 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 automata representation described in Section 5 is based on prior work on symbolic automata [14, 43, 51]. One notable difference with previous work is that the construction of automata in our setting is complicated by the fact that theories can introduce predicates that take into account the entire view of the previous trace. The separate account of theory and term automata we present takes this into account is based on ideas in 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 equational theory sound with respect to a trace semantics, and automatically construct automata-theoretic decision procedures for the KMT. Our theoretical framework captures common use cases in the form of theories for bitvectors, natural numbers, unbounded sets and maps, networks, and temporal logic.

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. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[2] M Anderson. Time warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.

[3] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical report, Cornell University, Ithaca, NY, USA, 2001.

[4] Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. *Theoretical Computer Science*, 155:291–319, 1995.

[5] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

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

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

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

[9] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. *SIGPLAN Not.*, 48 (1):457–468, January 2013.

[10] Eric Hayden Campbell. Infiniteness and linear temporal logic: Soundness, completeness, and decidability. Undergraduate thesis, Pomona College, 2017.

[11] Ernie Cohen. Using kleene algebra to reason about concurrency control. Technical report, Telcordia, 1994.

[12] Ernie Cohen. Hypotheses in kleene algebra, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.6067.

[13] Ernie Cohen. Lazy caching in kleene algebra, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5074.

[14] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. *SIGPLAN Not.*, 49(1):541–553, January 2014. ISSN 0362-1340.

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

[16] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on ltl on finite traces: Insensitivity to infiniteness. In *AAAI*, pages 1027–1033. Citeseer, 2014.

[17] Leonardo De Moura and Nikolaj Bjørner. 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[19] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8): 453–457, August 1975.

[20] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194 – 211, 1979.

[21] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 343–355, New York, NY, USA, 2015. ACM.

[22] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In Peter Thiemann, editor, *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*, pages 282–309, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[23] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *CoRR*, abs/0909.0884, 2009.

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

[25] Murdoch J. Gabbay and Vincenzo Ciancia. 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, pages 365–380, Berlin, Heidelberg, 2011.

[26] Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. Automating full functional verification of programs with loops. *CoRR*, abs/1407.5286, 2014. URL http://arxiv.org/abs/1407.5286.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[28] Joanne Godfrey. The summer of network misconfigurations. https://blog.algosec.com/2016/08/business-outages-caused-misconfigurations-headline-news-summer.html, 2016.

[29] Niels Bjørn Bugge Grathwohl, Dexter Kozen, and Konstantinos Mamouras. 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, pages 44:1–44:10, New York, NY, USA, 2014. ACM.

[30] Zeus Kerravala. What is behind network downtime? proactive steps to reduce human error and improve availability of networks. https://www.cs.princeton.edu/courses/archive/fall10/cos561/papers/Yankee04.pdf, 2004.

[31] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. 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, pages 328–343, 2010.

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

[33] Dexter Kozen. Some results in dynamic model theory. *Science of Computer Programming*, 51(1):3 – 22, 2004. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2003.09.004. URL http://www.sciencedirect.com/science/article/pii/S0167642304000115. Mathematics of Program Construction (MPC 2002).

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

[35] Dexter Kozen and Maria-Christina Patron. Certification of compiler optimizations using kleene algebra with tests. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 568–582, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67797-6.

[36] Kim G Larsen, Stefan Schmid, and Bingtian Xue. Wnetkat: Programming and verifying weighted software-defined networks. In *OPODIS*, 2016.

[37] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.

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

[39] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[40] Juniper Networks. As the value of enterprise networks escalates, so does the need for configuration management. https://www-935.ibm.com/services/au/gts/pdf/200249.pdf, 2008.

[41] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 42–56, New York, NY, USA, 2016.

[42] Rohit Parikh. The completeness of propositional dynamic logic. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978: Proceedings, 7th Symposium Zakopane, Poland, September 4–8, 1978*, pages 403–415, Berlin, Heidelberg, 1978.

[43] Damien Pous. 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, pages 357–368, New York, NY, USA, 2015.

[44] Yakov Rekhter and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 1654, RFC Editor, July 1995. URL http://www.rfc-editor.org/rfc/rfc1654.txt.

[45] J. J.M.M. Rutten. Universal coalgebra: A theory of systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1996.

[46] Andrew E. Santosa. Comparing weakest precondition and weakest liberal precondition. *CoRR*, abs/1512.04013, 2015.

[47] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 11–24, New York, NY, USA, 2014. ACM.

[48] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 88–105, New York, NY, USA, 2014.

[49] Simon Sharwood. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/, 2016.

[50] Alexandra Silva. Kleene coalgebra. Phd thesis, University of Minho, Braga, Portugal, 2010.

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

[52] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS*, 2001.

[53] Yevgenly Sverdlik. Microsoft: misconfigured network device led to azure outage. http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, 2012.

[54] Burghard von Karger. Temporal algebra. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 309–385. Springer-Verlag New York, Inc., New York, NY, USA, 2002.