Kleene Algebra Modulo Theories

A Framework for Concrete KATs

Michael Greenberg
Stevens Institute of Technology
Hoboken, NJ, USA
michael@greenberg.science

Ryan Beckett
Microsoft Research
Redmond, WA, USA
Ryan.Beckett@microsoft.com

Eric Campbell
Cornell University
Ithaca, NY, USA
ehc86@cornell.edu

Abstract

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

We offer a fast path to a "minimum viable model" of a KAT, formally or in code through our framework, *Kleene algebra modulo theories* (KMT). Given primitives and a notion of state, we can automatically derive a corresponding KAT's semantics, prove its equational theory sound and complete with respect to a tracing semantics (programs are denoted as traces of states), and derive a normalization-based decision procedure for equivalence checking. Our framework is based on *pushback*, a generalization of weakest preconditions that specifies how predicates and actions interact. We offer several case studies, showing tracing variants of theories from the literature (bitvectors, NetKAT) along with novel compositional theories (products, temporal logic, and sets). We derive new results over *unbounded state*, reasoning about monotonically increasing, unbounded natural numbers. Our OCaml implementation closely matches the theory: users define and compose KATs with the module system.

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; *Frameworks*; *Formal software verification*; *Correctness*; *Automated static analysis*; • **Theory of computation** → **Regular languages**.

ACM Reference Format:
Michael Greenberg, Ryan Beckett, and Eric Campbell. 2022. Kleene Algebra Modulo Theories: A Framework for Concrete KATs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523722

1 Introduction

Kleene algebra with tests (KAT) provides a powerful framework for reasoning about regularly structured programs. Modeling simple programs with while loops and beyond, KATs can handle a variety of analysis tasks [2, 7, 11–13, 36] and typically enjoy sound, complete, and decidable equational theories. Interest in KATs has followed their success in networking: NetKAT, a language for programming and verifying Software Defined Networks (SDNs), is a remarkably successful concrete KAT [1], followed by many other variations and extensions [4, 8, 22, 37, 38, 48].

What's holding back KAT and its decidable equivalence from applying in other domains? *It's hard to generate useful, concrete instances of KAT.* But defining concrete KATs remains a challenging task even for KAT experts. To build a custom KAT, one must craft custom domain primitives, derive a collection of new domain-specific axioms, prove the soundness and completeness of the resulting algebra, and implement a decision procedure. For example, NetKAT's theory and implementation was developed over several papers [1, 23, 51], following a series of papers that resembled, but did not use, the KAT framework [21, 29, 39, 44]. A pessimistic analysis concludes that making a domain-specific KAT requires moving to an institution with a KAT expert!

Abstract KAT has not successfully transferred to other domains. The conventional, abstract approach to KAT leaves actions and predicates abstract, without any domain-specific equations [15, 34, 40, 43]. Abstract KATs can't do domain-specific reasoning. Domain-specific knowledge must be encoded manually as additional equational assumptions, which makes equivalence undecidable in general; decision procedures have limited support for reasoning over domain-specific primitives and axioms [11, 32]. Applying KAT is hard because existing work is too abstract, with challengingly telegraphic completeness proofs: normalization procedures are implicit in the very terse proofs. Such concision makes it hard for a domain expert to adapt KAT to their needs.

Domain-specific KATs will find more general application when it is possible to cheaply build and experiment with them. Our goal is to democratize KATs, offering a general framework for automatically deriving sound, complete, and decidable KATs with tracing semantics for client theories.

By tracing semantics, we mean that programs are denoted as traces of actions and states. Such a semantics is useful for proving not just end-to-end properties (e.g., does the packet arrive at the correct host?) but also more fine-grained ones (e.g., does the packet traverse the firewall on the way?).

To demonstrate the effectiveness of our approach, we not only reproduce results from the literature (e.g., tracing variants of finite-state KATs, like bit vectors and NetKAT), but we also derive new KATs that have monotonically increasing, unbounded state (e.g., naturals). The proof obligations of our approach are relatively mild and our approach is *compositional*: a client can compose smaller theories to form larger, more interesting KATs than might be tractable by hand. Our completeness proof corresponds directly to a *modular* equivalence decision procedure; users compose KATs and their decision procedures from theories specified as OCaml modules. We offer a fast path to a "minimum viable model" for those wishing to experiment with KATs.

1.1 What is a KAT?

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

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

To model such a program in KAT, one replaces each concrete test or action with an abstract representation. Let the atomic test α represent the test $\mathtt{i} < 50$, β represent $\mathtt{i} < 100$, and γ represent $\mathtt{j} > 100$; the atomic actions p and q represent the assignments $\mathtt{i} := \mathtt{i} + 1$ and $\mathtt{j} := \mathtt{j} + 2$, respectively. We can now write the program as the KAT expression $\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg\beta \cdot \gamma$. The complete equational theory of KAT makes it possible to reason about program transformations and decide equivalence between KAT terms. For example, KAT's theory that the original loop is equivalent to its unfolding:

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

But there's a catch: α and β and p and q are abstract. KATs are naïvely propositional, with no model of the underlying domain or the semantics of the abstract predicates and actions. For example, the fact that $(\mathtt{j} := \mathtt{j} + 2 \cdot \mathtt{j} > 200) \equiv (\mathtt{j} > 198 \cdot \mathtt{j} := \mathtt{j} + 2)$ does not follow from the KAT axioms and must be added manually to any proof as an equational assumption. Yet the ability to reason about the equivalence of programs in the presence of particular domains is critical for reasoning about real programs and domain-specific languages. Unfortunately, it remains an expert's task to extend the KAT with new domain-specific axioms, provide new proofs of soundness and completeness, and develop the corresponding implementation [1, 4, 8, 28, 35].

As an example of such a domain-specific KAT, NetKAT models packet forwarding in computer networks as KAT terms. 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): a network is the logical crossbar $in; (p; t)^*; p; out$, where p is a policy, t models the network topology and in and out are edge predicates. NetKAT extends KAT with two actions and one predicate: an action to write to packet fields, $f \leftarrow v$, where we write value v to field f of the current packet; an action dup, which records a packet in a history log; and a field matching predicate, $f = v$, which determines whether the field f of the current packet is set to the value v. Each NetKAT program is denoted as a function from a packet history to a set of packet histories. For example, the program:

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

takes a packet history as input, updates the current packet to have a new destination IP address and port, and then records the current packet state. The original NetKAT paper defines a denotational semantics not just for its primitive parts, but for the various KAT operators; they explicitly restate the KAT equational theory along with custom axioms for the new primitive forms, prove the theory's soundness, and then devise a novel normalization routine to reduce NetKAT to an existing KAT with a known completeness result. Later papers [23, 51] then developed the NetKAT automata theory used to compile NetKAT programs into forwarding tables and to verify networks. NetKAT's power is costly: one must prove metatheorems and develop an implementation—a high bar to meet for those hoping to apply KAT in their domain.

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

[1] KATs are more general, though—guarded KAT [52] corresponds directly to While programs, while KAT admits general parallel composition and iteration.

Syntax

$\alpha ::= x > n \quad \pi ::= \mathsf{inc}_x \mid x := n \quad \mathsf{sub}(x > n) = \{x > m \mid m \leq n\}$

Semantics

$$n \in \mathbb{N} \qquad x \in \mathcal{V} \qquad \mathsf{State} = \mathcal{V} \to \mathbb{N}$$

$$\mathsf{pred}(x > n, t) = \mathsf{last}(t)(x) > n$$

$$\mathsf{act}(\mathsf{inc}_x, \sigma) = \sigma[x \mapsto \sigma(x) + 1] \qquad \mathsf{act}(x := n, \sigma) = \sigma[x \mapsto n]$$

Weakest precondition

$x := n \cdot (x > m)$ WP $(n > m)$

$\mathsf{inc}_y \cdot (x > n)$ WP $(x > n)$ $\qquad$ $\mathsf{inc}_x \cdot (x > 0)$ WP 1

$\mathsf{inc}_x \cdot (x > n)$ WP $(x > n - 1)$ when $n \neq 0$

Axioms

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

Figure 1. IncNat, increasing naturals

which we call Kleene algebras modulo theories (KMT) after the objects it produces, allows us to derive metatheory and implementation for KATs based on a given theory. The KMT framework obviates the need to deeply understand KAT metatheory and implementation for a large class of extensions; a variety of higher-order theories allow language designers to compose new KATs from existing ones, allowing them to rapidly prototype their KAT theories.

1.2 An example instance: incrementing naturals

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

Proof obligations. Our framework takes a *client theory* and produces a KAT, but what must one provide in order to know that the generated KAT is deductively complete, or to derive an implementation? We require, at a minimum, a description of the theory's *primitive* predicates and actions along with how these apply to some notion of state. We call these parts the *client theory* (Fig. 1). The resulting KAT is a *Kleene algebra modulo theory* (KMT).

Our framework hinges on an operation relating predicates and operations called *pushback*. Pushback is a generalization of weakest preconditions, built out of a notion of weakest preconditions for each pair of primitive test and action. Accordingly, client theories must define a weakest preconditions relation WP along with axioms that are sufficient to justify WP. The WP relation provides a way to compute the weakest precondition for any primitive action and test: we write $\pi \cdot \alpha$ WP a to mean that $\pi \cdot \alpha \equiv a \cdot \pi$. For example, the weakest precondition of $\mathsf{inc}_x \cdot x > n$ is $x > n-1$ when n is not zero; the weakest preconditoin of $x := n \cdot (x > m)$ is $n > m$, which is statically either 1 (when the constant n is greater than the constant m) or 0 (otherwise). The client theory's WP should have two properties: it should be sound, (i.e., the resulting expression is equivalent to the original one); and none of the resulting predicates should be any bigger than the original predicates, by some measure (see §3). For example, the domain axiom: $\mathsf{inc}_x \cdot (x > n) \equiv (x > n - 1) \cdot \mathsf{inc}_x$ ensures that weakest preconditions for inc_x are modeled by the equational theory. The other axioms are used to justify the remaining weakest preconditions that relate other actions and predicates. Additional axioms that do not involve actions (Gt-Contra, GT-Min), are included to ensure that the predicate fragment of IncNat is complete in isolation.

Formally, the client must provide the following for our normalization routine (part of completeness): primitive tests and actions (α and π), semantics for those primitives (states σ and functions pred and act), a function identifying each primitive's subterms (sub), a weakest precondition relation (WP) justified by sound domain axioms ($\equiv$), restrictions on WP term size growth. In addition to these definitions, our client theory incurs a few proof obligations: $\equiv$ must be sound with respect to the semantics; the pushback relation should never push back a term that's larger than the input; the pushback relation should be sound with respect to $\equiv$; and we need a satisfiability checking procedure for a Boolean algebra extended with the primitive predicates. Given these things, we can construct a sound and complete KAT with a normalization-based equivalence procedure. For this example, the deductive completeness of the model shown here can be reduced to Presburger arithmetic.

It was relatively easy to define IncNat, and we get real power—we've extended KAT with unbounded state. It is sound to add other operations to IncNat, like scalar multiplication or addition. So long as the operations are monotonically increasing and invertible, we can still define a WP and corresponding axioms. It is *not* possible, however, to compare two variables directly with tests like $x = y$—doing so would break the requirement that weakest precondition does not enlarge tests. Put another way, the test $x = y$ can encode context-free languages! The non-KMT term $x := 0 \cdot y := 0; (\mathsf{inc}_x)^* \cdot (\mathsf{inc}_y)^* \cdot x = y$ does balanced increments of x and y. For similar reasons, we cannot add a decrement operation dec_x. Either of these would let us define counter machines, leading inevitably to undecidability.

Implementation. Users implement KMT's client theories by defining OCaml modules; users give the types of actions and tests along with functions for parsing, computing subterms, calculating weakest preconditions for primitives, mapping predicates to an SMT solver, and deciding predicate satisfiability (see §4 for more detail).

Our example implementation starts by defining a new, recursive module called `IncNat`. Recursive modules let the client theory make use of the derived KAT functions and types. For example, the module `K` on the fifth line gives us a recursive reference to the resulting KMT instantiated with the `IncNat` theory; such self-reference is key for higher-order theories, which must embed KAT predicates inside of other kinds of predicates (§2). The client defines two types: tests a and actions p. Here, tests are just $x > n$ where variables are `strings`, and numbers are `ints`. Actions store the variable being incremented (inc_x); we omit assignment to save space.

```
type a = Gt of string * int   (* alpha ::= x > n *)
type p = Increment of string (* pi    ::= inc x *)
module rec IncNat : THEORY
 (* generated KMT, for recursive use *)
 module K = KAT (IncNat)
 (* extensible parser; pushback; subterms *)
 let parse name es = ...
 let push_back p a = match (p,a) with
  | (Increment x, Gt (y, j)) when x = y →
    singleton_set (K.theory (Gt (y, j − 1)))
  | ...
 let rec subterms x = ...
 (* decision procedure for predicates *)
 let satisfiable (a: K.Test.t) = ...
end
```

The first function, `parse`, allows the library author to extend the KAT parser (if desired) to include new kinds of tests and actions in terms of infix and named operators. The implementation obligations—syntactic extensions, subterms functions, WP on primitives, a satisfiability checker for the test fragment—mirror our formal development. We offer more client theories in §2 and more implementation detail in §4.

Contributions. We claim the following contributions:

- A compositional framework for defining KATs and proving their metatheory, with a novel development of the normalization procedure used in completeness (§3). Completeness yields a decision procedure based on normalization.
- Case studies of this framework (§2), several of which reproduce results from the literature, and several of which are new and cover unbounded state: base theories (e.g., naturals, bitvectors [28], networks), and more importantly, compositional, higher-order theories (e.g., sets and LTL_f). We derive Temporal NetKAT compositionally [8] by applying the theory of LTL_f to a theory of NetKAT; doing so strengthens Temporal NetKAT's completeness result.
- An implementation of KMT (§4) mirroring our proofs; deriving an equivalence decision procedure for client theories from just a few definitions. Our implementation is efficient enough to experiment with small programs (§5).

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

Notes for KAT experts. To restate our contributions for readers more deeply familiar with KAT: our work is quite different from conventional work on KAT, which tends to focus on abstract, general theories. KAT's success in NetKAT is portable, but would-be KAT users need help constructing concrete KAT instances. Our framework is similar to Schematic KAT [33]. But Schematic KAT is incomplete. Our framework identifies a complete subset of Schematic KATs: tracing semantics and monotonic weakest preconditions.

2 Case studies

We define KAT client theories for bitvectors and networks, as well as higher-order theories for products of theories, sets, and temporal logic (Fig. 2). To give a sense of the range and power of our framework, we offer these case studies before the formal details of the framework itself (§3).

2.1 Bit vectors

The simplest KMT is bit vectors: we extend KAT with some finite number of bits, each of which can be set to true or false and tested for their current value (Fig. 2(a)). The theory adds actions $b := \mathfrak{t}$ and $b := \mathfrak{f}$ for boolean variables b, and tests of the form $b = \mathfrak{t}$, 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: $b = \mathfrak{f}$ desugars to $\neg(b = \mathfrak{t})$; flip b desugars to $(b = \mathfrak{t} \cdot b := \mathfrak{f}) + (b = \mathfrak{f} \cdot b := \mathfrak{t})$.We could go further and define numeric operators on collections of bits, at the cost of producing larger terms. We are not limited to numbers, of course; once we have bits, we can encode any bounded structure we like. KAT+B! [28] develops a similar theory, but our semantics admit different equalities. KMT uses *trace* semantics, distinguishing $b := \mathfrak{t} \cdot b := \mathfrak{t}$ and $b := \mathfrak{t}$. 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 $(b := \mathfrak{t} \cdot b := \mathfrak{t}) \equiv (b := \mathfrak{t})$. KMT can't *exactly* model KAT+B!. (We have a similar relationship to NetKAT (§2.5).) It's difficult to say if one model is 'better'—either could be appropriate, depending on the setting. For example, our tracing semantics is useful for answering model-checking-like questions (§2.4).

Syntax

$$\alpha ::= b = \mathsf{t} \qquad \pi ::= b := \mathsf{t} \mid b := \mathsf{f}$$

$$\mathsf{sub}(\alpha) = \{\alpha\}$$

Semantics

$$b \in \mathcal{B} \qquad \mathsf{State} = \mathcal{B} \to \{\mathsf{t}, \mathsf{f}\}$$

$$\mathsf{pred}(b = \mathsf{t}, t) = \mathsf{last}(t)(b) \qquad \mathsf{act}(b := \mathsf{t}, \sigma) = \sigma[b \mapsto \mathsf{t}] \qquad \mathsf{act}(b := \mathsf{f}, \sigma) = \sigma[b \mapsto \mathsf{f}]$$

Weakest precondition

$$b := \mathsf{t} \cdot b = \mathsf{t} \ \mathsf{WP}\ 1 \qquad b := \mathsf{f} \cdot b = \mathsf{t}\ \mathsf{WP}\ 0$$

Axioms

$$(b := \mathsf{t}) \cdot (b = \mathsf{t}) \equiv (b := \mathsf{t}) \ \textsc{True-True} \qquad (b := \mathsf{f}) \cdot (b = \mathsf{t}) \equiv 0 \ \textsc{False-True}$$

(a) BitVec, theory of bitvectors

Syntax

$$\alpha ::= \alpha_1 \mid \alpha_2 \qquad \pi ::= \pi_1 \mid \pi_2$$

$$\mathsf{sub}(\alpha_i) = \mathsf{sub}_i(\alpha_i)$$

Semantics

$$\mathsf{State} = \mathsf{State}_1 \times \mathsf{State}_2$$

$$\mathsf{pred}(\alpha_i, t) = \mathsf{pred}_i(\alpha_i, t_i) \qquad \mathsf{act}(\pi_i, \sigma) = \sigma[\sigma_i \mapsto \mathsf{act}_i(\pi_i, \sigma_i)]$$

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

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

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

$$\pi_1 \cdot \alpha_2 \equiv \alpha_2 \cdot \pi_1 \ \ \textsc{L-R-Comm} \qquad \pi_2 \cdot \alpha_1 \equiv \alpha_1 \cdot \pi_2 \ \ \textsc{R-L-Comm}$$

(b) Prod($\mathcal{T}_1, \mathcal{T}_2$), products of two disjoint theories

Syntax

$$\alpha ::= \mathsf{in}(x, c) \mid e = c \mid \alpha_e \qquad \pi ::= \mathsf{add}(x, e) \mid \pi_e$$

$$\begin{aligned} \mathsf{sub}(\mathsf{in}(x,c)) &= \{\mathsf{in}(x,c)\} \cup \mathsf{sub}(\neg(e = c)) \\ \mathsf{sub}(e = c) &= \mathsf{sub}(e = c) \\ \mathsf{sub}(\alpha_e) &= \mathsf{sub}(\alpha_e) \end{aligned}$$

Semantics

$$c \in \mathcal{C} \qquad e \in \mathcal{E} \qquad x \in \mathcal{V} \qquad \mathsf{State} = (\mathcal{V} \to \mathcal{P}(\mathcal{C})) \times (\mathcal{E} \to \mathcal{C})$$

$$\mathsf{pred}(\mathsf{in}(x,c), t) = c \in \mathsf{last}(t)_1(x) \qquad \mathsf{pred}(\alpha_e, t) = \mathsf{pred}(\alpha_e, t_2)$$

$$\begin{aligned} \mathsf{act}(\mathsf{add}(x,e), \sigma) &= \sigma[\sigma_1[x \mapsto \sigma_1(x) \cup \{\sigma(e)\}]] \\ \mathsf{act}(\pi_e, \sigma) &= \sigma[\sigma_2 \mapsto \mathsf{act}(\pi_e, \sigma_2)] \end{aligned}$$

Weakest precondition extending $\mathcal{E}$

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

Axioms extending $\mathcal{E}$

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

(c) Set($\mathcal{E}$), unbounded sets over expressions

Syntax

$$\alpha ::= \bigcirc a \mid a \,\mathcal{S}\, b \mid a \qquad \pi ::= \pi_{\mathcal{T}}$$

$$\begin{aligned} \mathsf{sub}(\bigcirc a) &= \{\bigcirc a\} \cup \mathsf{sub}(a) \\ \mathsf{sub}(a \,\mathcal{S}\, b) &= \{a \,\mathcal{S}\, b\} \cup \mathsf{sub}(a) \cup \mathsf{sub}(b) \end{aligned}$$

$$\bullet\, a \triangleq \neg \bigcirc \neg a \qquad a \,\mathcal{B}\, b \triangleq a \,\mathcal{S}\, b + \Box a$$

$$\mathsf{start} \triangleq \neg \bigcirc 1 \qquad \Diamond a \triangleq 1 \,\mathcal{S}\, a \qquad \Box a \triangleq \neg \Diamond \neg a$$

Weakest precondition extending $\mathcal{T}$

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

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

$$a \leq \bullet\, a \cdot b \ \rightarrow \ a \leq \Box b \quad \textsc{LTL-Induction}$$

Semantics

$$\mathsf{State} = \mathsf{State}_{\mathcal{T}}$$

$$\mathsf{pred}(\bigcirc a, \langle \sigma, l \rangle) = \mathsf{f} \qquad \mathsf{pred}(\bigcirc a, t\langle \sigma, l \rangle) = \mathsf{pred}(a, t)$$

$$\begin{aligned} \mathsf{pred}(a \,\mathcal{S}\, b, \langle \sigma, l \rangle) &= \mathsf{pred}(b, \langle \sigma, l \rangle) \\ \mathsf{pred}(a \,\mathcal{S}\, b, t\langle \sigma, l \rangle) &= \mathsf{pred}(b, t\langle \sigma, l \rangle) \vee (\mathsf{pred}(a, t\langle \sigma, l \rangle) \wedge \mathsf{pred}(a \,\mathcal{S}\, b, t)) \end{aligned}$$

$$\mathsf{act}(\pi, \sigma) = \mathsf{act}(\pi, \sigma)$$

Axioms (extending those of $\mathcal{T}$)

$$\begin{aligned} \bigcirc(a \cdot b) &\equiv \bigcirc a \cdot \bigcirc b && \textsc{LTL-Last-Dist-Seq} \\ \bigcirc(a + b) &\equiv \bigcirc a + \bigcirc b && \textsc{LTL-Last-Dist-Plus} \\ \bullet\, 1 &\equiv 1 && \textsc{LTL-WLast-One} \\ a \,\mathcal{S}\, b &\equiv b + a \cdot \bigcirc(a \,\mathcal{S}\, b) && \textsc{LTL-Since-Unroll} \\ \neg(a \,\mathcal{S}\, b) &\equiv (\neg b) \,\mathcal{B}\, (\neg a \cdot \neg b) && \textsc{LTL-Not-Since} \\ \Box a &\leq \Diamond(\mathsf{start} \cdot a) && \textsc{LTL-Finite} \end{aligned}$$

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

Figure 2. Client theories for KMT; in higher-order theories, we highlight client obligations.

2.2 Disjoint products

Given two client theories, we can combine them into a disjoint product theory, Prod($\mathcal{T}_1, \mathcal{T}_2$), where the states are products of the two sub-theory's states and the predicates and actions from $\mathcal{T}_1$ can't affect $\mathcal{T}_2$ and vice versa (Fig. 2(b)). We explicitly give definitions for pred and act that defer to the corresponding sub-theory, using t_i to project the trace state to the ith component. It may seem that disjoint products don't give us much, but they in fact allow for us to simulate much more interesting languages in our derived KATs. For example, Prod(BitVec, IncNat) allow boolean- or (increasing) natural-valued variables; the product theory lets us directly express things Kozen [32] encoded manually, i.e., loops over boolean and numeric state.

2.3 Unbounded sets

We define a KMT for unbounded sets parameterized on a theory of expressions $\mathcal{E}$ (Fig. 2(c)). We also support maps, but we omit them out of space concerns. The set data type supports just one operation: $\mathsf{add}(x, e)$ adds the value of expression e to set x (we could add $\mathsf{del}(x, e)$, but we omit it to save space). It also supports a single test: $\mathsf{in}(x, c)$ checks if the constant c is contained in set x. The idea is that $e \in \mathcal{E}$ refers to expressions with, say, variables x and constants c. We allow arbitrary expressions e in some positions and constants c in others. It's critical that for each constant c, the test of expression equality $e = c$ be smaller in our global ordering than the membership test $\mathsf{in}(x, c)$. For example, we can have sets of naturals by setting $\mathcal{E} ::= n \in \mathbb{N} \mid i \in \mathcal{V}'$, where our constants $\mathcal{C} = \mathbb{N}$ and $\mathcal{V}'$ is some set of variables distinct from those we use for sets. We 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).

To instantiate the Set theory, we need a few things: expressions $\mathcal{E}$, a subset of *constants* $\mathcal{C} \subseteq \mathcal{E}$, and predicates for testing (in)equality between expressions and constants ($e = c$ and $e \neq c$). Comparing two variables would cause us to accidentally define a counter machine. Our state has two parts: $\sigma_1 : \mathcal{V} \to \mathcal{P}(\mathcal{C})$ records the current sets for each set in $\mathcal{V}$, while $\sigma_2 : \mathcal{E} \to \mathcal{C}$ evaluates expressions in each state. Since each state has its own evaluation function, the expression language can have actions that update σ_2.

2.4 Past-time linear temporal logic

Past-time linear temporal logic on finite traces (LTL_f) [5, 8–10, 16, 17, 45] is a *higher-order theory*: LTL_f extends another theory $\mathcal{T}$, with its own predicates and actions. Any $\mathcal{T}$ test can appear in of LTL_f's temporal predicates (Fig. 2(d)).

LTL_f adds just two predicates: $\bigcirc a$, pronounced "last a", means a held in the prior state; and $a \mathcal{S} b$, pronounced "a since b", means b held at some point in the past, and a has held since then. There is a slight subtlety around the beginning of time: we say that $\bigcirc a$ is false at the beginning (what can be true in a state that never happened?), and $a \mathcal{S} b$ degenerates to b at the beginning of time. These two predicates suffice to encode the rest of LTL_f; encodings are given below the syntax. Weakest preconditions uses inference rules: to push back $\mathcal{S}$, we unroll $a \mathcal{S} b$ into $a \cdot \bigcirc(a \mathcal{S} b) + b$; pushing last through an action is easy, but pushing back a or b recursively uses the $\mathrm{PB}^\bullet$ judgment from the normalization routine of the KMT framework (Fig. 5). Our implementation's recursive modules let us use the derived pushback to define weakest preconditions.

The equivalence axioms come from Temporal NetKAT [8]; the deductive completeness result for these axioms comes from Campbell's undergraduate thesis [9, 10]; Roşu's proof uses different axioms [45].

Syntax

$$\alpha ::= f = v \qquad \pi ::= f \leftarrow v \qquad \mathsf{sub}(\alpha) = \{\alpha\}$$

Semantics

$$\mathsf{F} = \text{packet fields} \qquad \mathsf{V} = \text{packet field values} \qquad \mathsf{State} = \mathsf{F} \to \mathsf{V}$$

$$\mathsf{pred}(f = v, t) = \mathsf{last}(t).f = v \qquad \mathsf{act}(f \leftarrow v, \sigma) = \sigma[f \mapsto v]$$

Weakest precondition

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

Axioms

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

Figure 3. Tracing NetKAT a/k/a NetKAT without dup

2.5 Tracing NetKAT

We define NetKAT as a KMT over packets, which we model as functions from packet fields to values (Fig. 3). KMT's tracing semantics diverge slightly from NetKAT's: like KAT+B! (§2.1; [28]), NetKAT normally merges adjacent writes. If the policy analysis demands reasoning about the history of packets traversing the network—reasoning, for example, about which routes packets actually take—the programmer must insert dups to record relevant moments in time. From our 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.

2.6 Temporal NetKAT

We derive Temporal NetKAT as $\mathrm{LTL}_f(\mathrm{NetKAT})$, i.e., LTL_f instantiated over tracing NetKAT; the combination yields precisely the system described in the Temporal NetKAT paper [8]. Recent proofs of deductive completeness for LTL_f [9, 10] yield a stronger completeness result—the original work showed completeness only for "network-wide" policies, i.e., those with start at the front.

3 The KMT framework

The rest of our paper describes how our framework takes a client theory and generates a KAT. We emphasize that you need not understand the following formalism to use our framework—we do it once and for all, so you don't have to! In figures, we highlight what the client theory must provide.

We derive a KAT $\mathcal{T}^*$ (Fig. 4) from a client theory $\mathcal{T}$, where $\mathcal{T}$ has two *primitive* parts—predicates $\alpha \in \mathcal{T}_\alpha$ and actions

$\pi \in \mathcal{T}_\pi$. Lifting $\mathcal{T}_\alpha$ to a Boolean algebra yields $\mathcal{T}^*_{\mathsf{pred}} \subseteq \mathcal{T}^*$, where $\mathcal{T}^*$ is the KAT that embeds the client theory.

A client theory must provide: (1) primitives α and π; (2) a notion of state and semantics for those primitives on that state; (3) theory-specific axioms of KAT equivalences that should hold, in terms of α and π ($\equiv_\mathcal{T}$); (4) a weakest precondition operation WP that relates each α and π; and (5) a satisfiability checker for the theory's predicates, i.e., for $\mathcal{T}^*_{\mathsf{pred}}$. (See §4 for details on how these are provided.)

Our framework provides results for $\mathcal{T}^*$ in a pay-as-you-go fashion: given just the state and an interpretation for the predicates and actions of $\mathcal{T}$, we derive a tracing semantics for $\mathcal{T}^*$ (§3.1); if the axioms of $\mathcal{T}$ are sound with respect to the tracing semantics, then $\mathcal{T}^*$ is sound (§3.2); if the axioms of $\mathcal{T}$ are complete with respect to our semantics and WP satisfies some ordering requirements, then $\mathcal{T}^*$ has a complete equational theory (§3.4); and finally, with just a bit of code defining the structure of $\mathcal{T}$ and deciding the predicate theory $\mathcal{T}^*_{\mathsf{pred}}$, we can derive a decision procedure for equivalence (§4) using the normalization routine from completeness (§3.4).

The key to our general, parameterized proof is a novel *pushback* operation that generalizes weakest preconditions (§3.3.2): given an understanding of how to push primitive predicates back to the front of a term, we can normalize terms for our completeness proof (§3.4).

3.1 Semantics

The first step in turning the client theory $\mathcal{T}$ into a KAT is to define a semantics (Fig. 4). We can give any KAT a *tracing semantics*: the meaning of a term is a trace t, which is a non-empty list of log entries l. Each *log entry* records a state σ and (in all but the initial state) a primitive action π. The client assigns meaning to predicates and actions by defining a set of states State and two functions: one to determine whether a predicate holds (pred) and another to determine an action's effects (act). To run a $\mathcal{T}^*$ term on a state σ, we start with an initial state $\langle \sigma, \bot \rangle$; when we're done, we'll have a set of traces of the form $\langle \sigma_0, \bot \rangle \langle \sigma_1, \pi_1 \rangle \dots$, where $\sigma_i = \mathsf{act}(\pi_i, \sigma_{i-1})$ for $i > 0$. (A similar semantics shows up in Kozen's application of KAT to static analysis [32].)

The client's pred function takes a primitive predicate α and a trace — predicates can examine the entire trace — returning true or false. When the pred function returns $\mathfrak{t}$, we return the singleton set holding our input trace; when pred returns $\mathfrak{f}$, we return the empty set. It's acceptable for pred to recursively call the denotational semantics (e.g., §2.6), though we have skipped the formal detail here.

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 π.

3.2 Soundness

Proving the equational theory sound relative to our tracing semantics is easy: we depend on the client's act and pred functions, and none of our KAT axioms refer to primitives (Fig. 4). Our soundness proof requires that the client theory's equations be sound in our tracing semantics.

Theorem 3.1 (Soundness of $\mathcal{T}^*$ relative to $\mathcal{T}$). *If* $p \equiv_\mathcal{T} q \Rightarrow [\![p]\!] = [\![q]\!]$ *then* $p \equiv q \Rightarrow [\![p]\!] = [\![q]\!]$.

Proof. By induction on the derivation of $p \equiv q$. □

If the client theory is buggy, i.e., the axioms are unsound, then we can offer no guarantees about $\mathcal{T}$ at all. For the duration of §3, we assume that any equations $\mathcal{T}$ adds are sound and, so, $\mathcal{T}^*$ is sound by Theorem 3.1.

3.3 Normalization via pushback

In order to prove completeness (§3.4), we reduce our KAT terms to a more manageable subset of *normal forms*. Normalization happens via a generalization of weakest preconditions; we use a *pushback* operation to translate a term p into an equivalent term of the form $\sum a_i \cdot m_i$ where each m_i does not contain any tests. The client theory's completeness result on the a_i then reduces the completeness of our language to an existing result for Kleene algebra on the m_i.

The client theory $\mathcal{T}$ must provide two things for our normalization procedure: (1) a way to extract subterms from predicates, which orders predicates for the termination measure on normalization (§3.3.1); and (2) weakest preconditions for primitives (§3.3.2). Once we've defined our normalization procedure, we can use it prove completeness (§3.4).

3.3.1 Normalization and the maximal subterm ordering. Our normalization algorithm works by "pushing back" predicates to the front of a term until we reach a normal form with *all* predicates at the front. The pushback algorithm's termination measure is complex: pushing a predicate back may not eliminate it; pushing test a back through π may yield $\sum a_i \cdot \pi$ where each of the a_i copies some subterm of a—and there may be *many* such copies!

Let the set of *restricted actions* $\mathcal{T}_{\mathsf{RA}}$ be the subset of $\mathcal{T}^*$ where the only test is 1. Let the 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 $m_i \in \mathcal{T}_{\mathsf{RA}}$. Let the metavariables t, u, v, w, x, y, and z to denote elements of $\mathcal{T}^*_{\mathsf{nf}}$; we typically write these sets as sums, i.e., $x = \sum_{i=1}^{k} a_i \cdot m_i$ means $x = \{(a_1, m_1), (a_2, m_2), \dots, (a_k, m_k)\}$. The sum notation is convenient, but normal forms must 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 $\mathcal{T}^*_{\mathsf{nf}}$.

Predicate syntax

$$\begin{array}{rcll} a, b & ::= & 0 & \textit{additive identity} \\ & \mid & 1 & \textit{multiplicative identity} \\ & \mid & \neg a & \textit{negation} \\ & \mid & a + b & \textit{disjunction} \\ & \mid & a \cdot b & \textit{conjunction} \\ & \mid & \alpha & \textit{primitive predicates } (\mathcal{T}_\alpha) \end{array}$$

Action syntax

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

Trace definitions

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

$$\begin{array}{lcl} \mathsf{pred} & : & \mathcal{T}_\alpha \times \mathsf{Trace} \to \{\mathsf{t}, \mathsf{f}\} \\ \mathsf{act} & : & \mathcal{T}_\pi \times \mathsf{State} \to \mathsf{State} \end{array}$$

Tracing semantics

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

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

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

Axioms (KA = Kleene algebra; BA = Boolean algebra)

$$\begin{array}{rl} p + (q + r) \equiv (p + q) + r & \textsc{KA-Plus-Assoc} \\ p + q \equiv q + p & \textsc{KA-Plus-Comm} \\ p + 0 \equiv p & \textsc{KA-Plus-Zero} \\ p + p \equiv p & \textsc{KA-Plus-Idem} \\ p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r & \textsc{KA-Seq-Assoc} \\ 1 \cdot p \equiv p & \textsc{KA-Seq-One} \\ p \cdot 1 \equiv p & \textsc{KA-One-Seq} \\ p \cdot (q + r) \equiv p \cdot q + p \cdot r & \textsc{KA-Dist-L} \\ (p + q) \cdot r \equiv p \cdot r + q \cdot r & \textsc{KA-Dist-R} \\ 0 \cdot p \equiv 0 & \textsc{KA-Zero-Seq} \\ p \cdot 0 \equiv 0 & \textsc{KA-Seq-Zero} \\ 1 + p \cdot p^* \equiv p* & \textsc{KA-Unroll-L} \\ 1 + p^* \cdot p \equiv p* & \textsc{KA-Unroll-R} \\ q + p \cdot r \le r \;\to\; p^* \cdot q \le r & \text{KA-LFP-L} \\ p + q \cdot r \le q \;\to\; p \cdot r^* \le q & \text{KA-LFP-R} \end{array}$$

$$\begin{array}{rl} a + (b \cdot c) \equiv (a + b) \cdot (a + c) & \textsc{BA-Plus-Dist} \\ a + 1 \equiv 1 & \textsc{BA-Plus-One} \\ a + \neg a \equiv 1 & \textsc{BA-Excl-Mid} \\ a \cdot b \equiv b \cdot a & \textsc{BA-Seq-Comm} \\ a \cdot \neg a \equiv 0 & \textsc{BA-Contra} \\ a \cdot a \equiv a & \textsc{BA-Seq-Idem} \end{array}$$

Consequences

$$\begin{array}{rl} (p + q)^* \equiv p^* \cdot (q \cdot p^*)^* & \textsc{Denesting} \\ p \cdot a \equiv b \cdot p \;\leftrightarrow\; p \cdot \neg a \equiv \neg b \cdot p & \textsc{Pushback-Neg} \\ p \cdot (q \cdot p)^* \equiv (p \cdot q)^* \cdot p & \textsc{Sliding} \\ p \cdot a \equiv a \cdot q + r \to p^* \cdot a \equiv (a + p^* \cdot r) \cdot q^* & \textsc{Star-Inv} \\ p \cdot a \equiv a \cdot q + r \to p \cdot a \cdot (p \cdot a)^* \equiv (a \cdot q + r) \cdot (q + r)^* & \textsc{Star-Expand} \end{array}$$

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

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

Our normalization algorithm uses the *maximal subterm ordering* as its termination measure. Here we simply give intuition for the two relevant high-level operations: $\mathsf{mt}(x) \subseteq \mathcal{T}^*_{\mathsf{pred}}$ computes the *maximal tests* of a normal form x, which are those tests that are not subterms of any other test. The maximal subterm ordering $x \preceq y$ for normal forms holds when the x's tests' subterms are a subset of y's tests' subterms. Informally, we have $x \preceq y$ when every test in x is somehow "covered" by a test in y; we have $x \prec y$ when $x \preceq y$ and y has some test x that does not. Our definition of subterms asks the client theory to identify the parts of its primitives via a function $\mathsf{sub}_\mathcal{T}$ such that (1) if $b \in \mathsf{sub}_\mathcal{T}(a)$ then $\mathsf{sub}(b) \subseteq \mathsf{sub}_\mathcal{T}(a)$ and (2) if $b \in \mathsf{sub}_\mathcal{T}(a)$, then either $b \in \{0, 1, a\}$ or b precedes a in a global ordering of predicates. We use the subterm ordering to shows we can always 'split' a normal form x around a maximal test $a \in \mathsf{mt}(x)$ 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. Splitting helps push tests back: the maximal test a (1) is factored out to the front of y and (2) does not appear in z at all.

Lemma 3.2 (Splitting)**.** *If $a \in \mathsf{mt}(x)$, then there exist y and z such that $x \equiv a \cdot y + z$ and $y \prec x$ and $z \prec x$.*

3.3.2 Pushback.

Normalization requires that the client theory's *weakest preconditions* respect the subterm ordering.

Definition 3.3 (Weakest preconditions)**.** The client theory's *weakest precondition* operation is a relation $\mathsf{WP} \subseteq \mathcal{T}_\pi \times \mathcal{T}_\alpha \times \mathcal{P}(\mathcal{T}^*_{\mathsf{pred}})$, where $\mathcal{T}_\pi$ are the primitive actions and $\mathcal{T}_\alpha$ are the primitive predicates of $\mathcal{T}$. WP need not be a function, but we require $\forall \pi \alpha \exists A, (\pi, \alpha, A) \in \mathsf{WP}$. We write $\pi \cdot \alpha \;\mathsf{WP}\; \sum a_i \cdot \pi$ and read it as "α pushes back through π to yield $\sum a_i \cdot \pi$" (the second π is purely notational). We require that if $\pi \cdot \alpha$ WP $\{a_1, \ldots, a_k\} \cdot \pi$, then $\pi \cdot \alpha \equiv \sum_{i=1}^{k} a_i \cdot \pi$, and $a_i \leq \alpha$.

Given the client theory's weakest-precondition relation WP, we define a normalization procedure for $\mathcal{T}^*$ by extending the client's WP relation to a more general *pushback* relation, PB (Fig. 5). The client's WP relation need not be a function, nor do the a_i need to be obviously related to α or π in any way. Even when the WP relation is a function, the PB relation generally won't be. While WP computes the classical weakest precondition, the PB relations are different: when pushing back we *change the program itself*—not normally an option for weakest preconditions (see §6).

The top-level normalization routine is the syntax-directed p norm x relation (Fig. 5), which 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 have single-summand normal forms where the predicate is 1 (Act); parallel composition of two normal forms means just joining the sums (Par). But sequence and Kleene star are harder: we define judgments using PB to lift these operations to normal forms (Seq, Star).

For sequences, we can recursively take $p \cdot q$ and normalize p into $x = \sum a_i \cdot m_i$ and q into $y = \sum b_j \cdot n_j$. To combine x and y, we can concatenate and rearrange the normal forms to get $\sum_{i,j} a_i \cdot m_i \cdot b_j \cdot n_j$. We write $x \cdot y$ PB^{J} z to mean that the concatenation of x and y is equivalent to the normal form z—the $\cdot$ is suggestive notation, here and elsewhere.

For Kleene star, we can take p^* and normalize p into $x = \sum a_i \cdot m_i$, but x^* isn't a normal form—we need to somehow move all of the tests out of the star and to the front. We do so with the PB^* relation (Fig. 5), writing x^* PB^* y to mean that the Kleene star of x is equivalent to the normal form y—the * on the left is again suggestive notation. The PB^* relation is more subtle than PB^{J}. Depending on how x splits (Lemma 3.2), there are four possibilities: if $x = 0$, then $0^* \equiv 1$ (StarZero); if x splits into $a \cdot x'$, then we can either use the KAT sliding lemma to pull the test out when a is strictly the largest test in x (Slide) or by using the KAT expansion lemma (Expand); if x splits into $a \cdot x' + z$, we use the KAT denesting lemma to pull a out before continuing recursively (Denest). SeqStarSmaller and SeqStarInv push a test (a) back through a star (m^*). Both rules work by unrolling the loop. In the simple case (SeqStarSmaller), the resulting test at the front is strictly smaller than a, and we can generate a normal form directly. When pushing the test back doesn't shrink a, we use Star-Inv to divide the term into parts with the maximal test ($a \cdot t$) and without it (u).

The bulk of the pushback's work happens in the $\mathsf{PB}^\bullet$ relation, which pushes a test back through a restricted action; PB^{R} and PB^{T} use $\mathsf{PB}^\bullet$ to push tests back through other forms. To handle negation, the function nnf—elided for space—puts predicates in *negation normal form*, where negations only appear on primitive predicates, using De Morgan's laws.

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 3.4). Then we show that the mutually recursive tangle of our PB relations always terminates (Theorem 3.5) .

Theorem 3.4 (Pushback soundness)**.** *Each of* PB *relations' left is equivalent to its right, e.g., if* x^* PB^* y *then* $x^* \equiv y$.

Proof. By simultaneous induction on the derivations. □

Finally, we show that every left-hand side of each pushback relation has a corresponding right-hand side. We *haven't* proved that the pushback relation is functional—there could be many different choices of maximal tests to push back.

Theorem 3.5 (Pushback existence)**.** *Each* PB *relations' left relates to some right that is no larger than the left's parts, e.g., for all* x *there exists* $y \leq x$ *such that* x^* PB^* y.

Proof. By induction on the lexicographical order of: the subterm ordering ($\prec$); the size of x; the size of m (for $\mathsf{PB}^\bullet$ and PB^{R}); and the size of a (for $\mathsf{PB}^\bullet$ and PB^{T}). Cases first split (Lemma 3.2) to show that derivations exist; subterm ordering congruence finds orderings to apply the IH. □

With pushback in hand, we show that every term has an equivalent normal form.

Corollary 3.6 (Normal forms)**.** *For all* $p \in \mathcal{T}^*$*, there exists a normal form* x *such that* p norm x *and that* $p \equiv x$.

Proof. By induction on p, using Theorems 3.5 and 3.4 in the Seq and Star case. □

The PB relations and these two proofs are one of the contributions of this paper: it is the first time that a KAT normalization procedure has been given as a distinct procedure, rather than hiding inside of normal forms used in completeness proofs. Temporal NetKAT, which introduced pushback, proved Theorems 3.4 and 3.5 as a single theorem, without any explicit normalization or pushback relation.

If the client's WP doesn't obey the axioms, then normalization may produce garbage. If the client's WP is sound but disrespects the global ordering, then normalization may not terminate, but any results it does produce will be correct.

3.4 Completeness

We prove completeness relative to our tracing semantics—if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$—by normalizing p and q and comparing the resulting terms. Like other completeness proofs, ours uses the completeness of Kleene algebra (KA) as its foundation: the set of possible traces of actions performed for a restricted (test-free) action in our denotational semantics is a regular language, and so the KA axioms are sound and complete for it. In order to relate our denotational semantics to regular languages, we define the regular interpretation of

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

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

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

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

$$\frac{m_i \cdot b_j \text{ PB}^{\bullet}\ x_{ij}}{(\sum_i a_i \cdot m_i) \cdot (\sum_j b_j \cdot n_j) \text{ PB}^{\text{J}} \sum_i \sum_j a_i \cdot x_{ij} \cdot n_j}\ \textsc{Join} \qquad \frac{}{m \cdot 0 \text{ PB}^{\bullet}\ 0}\ \textsc{SeqZero} \qquad \frac{}{m \cdot 1 \text{ PB}^{\bullet}\ 1 \cdot m}\ \textsc{SeqOne}$$

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

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

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

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

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

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

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

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

Figure 5. Normalization for $\mathcal{T}^*$

restricted actions $m \in \mathcal{T}_{\text{RA}}$ in the conventional way and then relate our denotational semantics to the regular interpretation. Our normalization routine only uses the KAT axioms and doesn't rely on any property of our tracing semantics. We conjecture that one could prove a similar completeness result and derive a similar decision procedure with a merging, non-tracing semantics, like in NetKAT or KAT+B! [1, 28].

We use several KAT theorems in our completeness proof (Fig. 4, Consequences), the most complex being star expansion (Star-Expand) [8]. Pushback-Neg is a novel generalization of a theorem of Cohen and Kozen [14, 31].

Theorem 3.7 (Completeness). *If the emptiness of $\mathcal{T}$ predicates is decidable, then if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$.*

Proof. There must exist normal forms x and y such that p norm x and q norm y and $p \equiv x$ and $q \equiv y$ (Corollary 3.6); 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 show that $x \equiv y$ to transitively prove that $p \equiv q$. We have $x = \sum_i a_i \cdot m_i$ and $y = \sum_j b_j \cdot n_j$. In principle, we ought to be able to match up each of the a_i with one of the b_j and then check to see whether m_i is equivalent to n_j (by appealing to the completeness on Kleene algebra). But we can't simply do a syntactic matching—we could have a_i and b_j that are in effect equivalent, but not obviously so. Worse still, we could have a_i and $a_{i'}$ equivalent! We need to perform two steps of disambiguation: first each normal form's predicates must be unambiguous locally, and then the predicates must be pairwise comparable between the two normal forms.

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

gets run precisely when a_i is true:

$$\left.\begin{array}{rl} \hat{x} = & a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot (m_1 + m_2 + \ldots + m_n) \\ + & \neg a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot (m_2 + \ldots + m_n) \\ + & a_1 \cdot \neg a_2 \cdot \ldots \cdot a_n \cdot (m_1 + \ldots + m_n) \\ + & \ldots \\ + & \neg a_1 \cdot \neg a_2 \cdot \ldots \cdot a_n \cdot m_n \\ + & \neg a_1 \cdot \neg a_2 \cdot \ldots \cdot \neg a_n \cdot 0 \end{array}\right\} \begin{array}{c} 2^n \\ \text{terms} \end{array}$$

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

Observe that the sum of all of the predicates in $\hat{x}$ and $\hat{y}$ are respectively equivalent to 1, since it enumerates all possible combinations of each a_i (BA-Plus-Dist, BA-Excl-Mid); i.e., if $\hat{x} = \sum_i c_i \cdot l_i$ and $\hat{y} = \sum_j d_j \cdot m_j$, then $\sum_i c_i \equiv 1$ and $\sum_j d_j \equiv 1$. We can take advantage of exhaustiveness of these sums to translate the locally disjoint but syntactically unequal predicates in each $\hat{x}$ and $\hat{y}$ to a single set of predicates on both, which allows us to do a *syntactic* comparison on each of the predicates. Let $\ddot{x}$ and $\ddot{y}$ be the extension of $\hat{x}$ and $\hat{y}$ with the tests from the other form, giving us $\ddot{x} = \sum_{i,j} c_i \cdot d_j \cdot l_i$ and $\ddot{y} = \sum_{i,j} c_i \cdot d_j \cdot m_j$. Extending the normal forms to be disjoint between the two normal forms is still provably equivalent using commutativity (BA-Seq-Comm) and the exhaustiveness above (KA-Seq-One).

Now that each of the predicates are syntactically uniform and disjoint, we can proceed to compare the commands. But there is one final risk: what if the $c_i \cdot d_j \equiv 0$? Then l_i and m_j could safely be different. Since we can check predicates of $\mathcal{T}$ for emptiness, we can eliminate those cases where the expanded tests at the front of $\ddot{x}$ and $\ddot{y}$ are equivalent to zero, which is sound by the client theory's completeness and zero-cancellation (KA-Zero-Seq and KA-Seq-Zero). If one normal form is empty, the other one must be empty as well.

Finally, we can defer to deductive completeness for KA to find proofs that the commands are equivalent. To use KA's completeness to get a proof over commands, we have to show that if our commands have equal denotations in our semantics, then they will also have equal denotations in the KA semantics. We've done exactly this by showing that restricted actions have regular interpretations: because the zero-canceled $\ddot{x}$ and $\ddot{y}$ are provably equivalent, soundness guarantees that their denotations are equal. Since their tests are pairwise disjoint, if their denotations are equal, it must be that any non-canceled commands are equal, which means that interpreting the commands as Kleene algebra (KA) terms over actions yields equal terms. By the deductive completeness of KA, we know that KA $\vdash l_i \equiv m_j$. Since $\mathcal{T}^*$ includes the KA axioms, $l_i \equiv m_j$; we have $c_i \cdot d_j \equiv c_i \cdot d_j$ by reflexivity, and so $\ddot{x} \equiv \ddot{y}$. By transitivity, we can see that $\hat{x} \equiv \hat{y}$ and so $x \equiv y$ and—finally!—$p \equiv q$. □

Our completeness proof relies on the client theory's decision procedure for satisfiability of $\mathcal{T}^*_{\text{pred}}$ terms. If the client theory's axioms are incomplete or this decision procedure is buggy, then the derived completeness proof may not be correct. With a broken decision procedure, the terms $\hat{x}/\hat{y}$ and $\ddot{x}/\ddot{y}$ might not actually be unambiguous, and so the output of the decision procedure would be garbage.

4 Implementation

Our formalism corresponds directly to our OCaml library.[2] Implementing a client theory means defining a module with the THEORY signature, lightly abridged:

```
module type THEORY = sig
 module A : CollectionType (* predicates *)
 module P : CollectionType (* actions *)
 (* recursive knot for KAT from A and P *)
 module Test with type t = A.t pred
 module Term with type t = (A.t, P.t) kat
 module K : KAT_IMPL
  with module A = A and module P = P
   and module Test = Test and module Term = Term
 (* lightweight extension to parser *)
 val parse : string → expr list → (A.t, P.t) either
 (* WP relation *)
 val push_back : P.t → A.t → Test.t set
 (* ordering *)
 val subterms : A.t → Test.t set
 (* optional routines for optimization *)
 val simplify_not : A.t → Test.t option
 val simplify_and : A.t → A.t → Test.t option
 val simplify_or : A.t → A.t → Test.t option
 val merge : P.t → P.t → P.t
 val reduce : A.t → P.t → P.t option
 (* satisfiability checker and z3 encoding *)
 val satisfiable : Test.t → bool
 val variable : P.t → string
 val variable_test : A.t → string
 val create_z3_var : string * A.t → Z3.context →
           Z3.Solver.solver → Z3.Expr.expr
 val theory_to_z3_expr : A.t → Z3.context →
            Z3.Expr.expr StrMap.t → Z3.Expr.expr
end
```

§1.2 summarizes the high-level idea and sketches a library implementation for the theory of increasing natural numbers. To use a higher-order theory like products, one need only instantiate the appropriate modules in the library:

```
module P = Product(IncNat)(Boolean)
module D = Decide(P)
let a = P.K.parse "y<1;(a=F + a=T; inc(y));y>0" in
let b = P.K.parse "y<1;a=T;inc(y)" in
```

[2] https://github.com/mgree/kmt

```
assert (D.equivalent a b)
```

The module `P` instantiates `Product` over our theories of incrementing naturals and booleans; the module `D` gives a way to normalize terms based on the completeness proof: it defines the normalization procedure along with the decision procedure `equivalent`. Users of the library can combine these modules to perform any number of tasks such as compilation, verification, inference, and so on. For example, checking language equivalence is then simply a matter of reading in KMT terms and calling the normalization-based equivalence checker. Our command-line tool works with these theories; given KMT terms in some supported theory as input, it partitions them into equivalence classes.

Our implementation uses several optimizations. The three most prominent are (1) hash-consing all KAT terms to ensure fast set operations, (2) lazy construction and exploration of word automata when checking actions for equivalence, and (3) domain-specific satisfiability checking for some theories.

5 Evaluation

We evaluated KMT on a collection of microbenchmarks exercising concrete KAT features (Fig. 6). For example, the second-to-last example does population count in a theory combining naturals and booleans: if a counter y is above a certain threshold, then the booleans a, b, and c must have been set to true. Our tool is usable for exploration—enough to decide whether to pursue any particular KAT.

Our normalization-based decision procedure is very fast in many cases. This is likely due to a combination of hash-consing and smart constructors that rewrite complex terms into simpler ones when possible, and the fact that, unlike previous KAT-based normalization proofs (e.g., [1, 32]) our normalization proof does not require splitting predicates into all possible "complete tests." However, our decision procedure does very poorly on examples where there is a sum nested inside of a Kleene star, i.e., $(p+q)^*$. The final, bit-flipping benchmark is one such example—it flips three boolean variables in some arbitrary order. In this case the normalization-based decision procedure repeatedly invokes the Denest rewriting rule, which greatly increases the size of the term on each invocation. Consider the simpler loop, which only flips from false to true: $(x_1 = \mathsf{f}; x_1 := \mathsf{t} + \cdots + x_n = \mathsf{f}; x_n := \mathsf{t})^*$. With $n = 1$, there are 4 disjunctions in the locally unambiguous form; with $n = 2$, there are 16; with $n = 3$, there are 512; with $n = 4$, there are 65,536. The normal forms grow in $O(2^{2^n})$, which quickly becomes intractable in space and time.

6 Related work

Kozen and Mamouras's Kleene algebra with equations [35] is perhaps the most closely related work: they also devise a framework for proving extensions of KAT sound and complete. Our works share a similar genesis: Kleene algebra with equations generalizes the NetKAT completeness proof (and then reconstructs it); our work generalizes the Temporal NetKAT completeness proof (and then reconstructs it—while also developing several other, novel KATs). Both their work and ours use rewriting to find normal forms and prove deductive completeness. Their rewriting systems work on mixed sequences of actions and predicates, but they can only delete these sequences wholesale or replace them with a single primitive action or predicate; our rewriting system's pushback operation only works on predicates (since the tracing semantics preserves the order of actions), but pushback isn't restricted to producing at most a single primitive predicate. Each framework can do things the other cannot. Kozen and Mamouras can accommodate equations that combine actions, like those that eliminate redundant writes in KAT+B! and NetKAT [1, 28]; we can accommodate more complex predicates and their interaction with actions, like those found in Temporal NetKAT [8] or those produced by the compositional theories (§2). A tracing semantics occurs in previous work on KAT as well [26, 32]. Selective tracing (à la NetKAT's dup) offers more control over which traces are considered equivalent; our pushback offers more flexibility for how actions and predicates interact. It may be possible to build a hybrid framework, with ideas from both.

Kozen studies KATs with arbitrary equations $x := e$ [33], also called Schematic KAT, where e comes from arbitrary first-order structures over a fixed signature Σ. He has a pushback-like axiom $x := e \cdot p \equiv p[{}^{x}/_{e}] \cdot x := e$. Arbitrary first-order structures over Σ's theory are much more expressive than anything we can handle—the pushback may or may not decrease in size, depending on Σ; KATs over such theories are generally undecidable. We, on the other hand, are able to offer pay-as-you-go results for soundness and completeness as well as an implementations for deciding equivalence—but only for first-order structures that admit a non-increasing weakest precondition. Other extensions of KAT often give up on decidabililty, too. Larsen et al. [37] allow comparison of variables, leading to an incomplete theory. They are, able, however, to decide emptiness of an entire expression.

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

Symkat is a powerful decision procedure for symbolic KAT, but doesn't work in our concrete setting [43]. It's possible to give symkat extra equations, and it can solve some equivalences that KMT can, but it can't handle, e.g., commutativity in general. Knotical uses KAT to model program traces for

Benchmark	$\mathcal{T}$	**Time to check equivalence**
$a^* \not\equiv a$ (for random arithmetic predicate a)	$\mathbb{N}$	0.034s
$\mathsf{inc}_x^*; x > 10 \equiv \mathsf{inc}_x^*; \mathsf{inc}_x^*; x > 10$	$\mathbb{N}$	<0.001s
$\mathsf{inc}_x^*; x > 3; \mathsf{inc}_y^*; y > 3 \equiv \mathsf{inc}_x^*; \mathsf{inc}_y^*; x > 3; y > 3$	$\mathbb{N}$	<0.001s
$x = \mathfrak{f}; (\mathsf{flip}\ x; \mathsf{flip}\ x)^* \equiv (\mathsf{flip}\ x; \mathsf{flip}\ x)^*; x = \mathfrak{f}$	$\mathcal{B}$	<0.001s
$w := \mathfrak{f}; x := \mathfrak{t}; y := \mathfrak{f}; z := \mathfrak{f};$ $(\mathsf{if}\ (w = \mathfrak{t} + x = \mathfrak{t} + y = \mathfrak{t} + z = \mathfrak{t})\ \mathsf{then}\ a := \mathfrak{t}\ \mathsf{else}\ a := \mathfrak{f})$ $\equiv\ w := \mathfrak{f}; x := \mathfrak{t}; y := \mathfrak{f}; z := \mathfrak{f};$ $(\mathsf{if}\ (w = \mathfrak{t} + x = \mathfrak{t}) + (y = \mathfrak{t} + z = \mathfrak{t})\ \mathsf{then}\ a := \mathfrak{t}\ \mathsf{else}\ a := \mathfrak{f})$	$\mathcal{B}$	<0.001s
$y < 1; a = \mathfrak{t}; \mathsf{inc}_y; (1 + b = \mathfrak{t}; \mathsf{inc}_y); (1 + c = \mathfrak{t}; \mathsf{inc}_y); y > 2$ $\equiv\ y < 1; a = \mathfrak{t}; b = \mathfrak{t}; c = \mathfrak{t}; \mathsf{inc}_y; \mathsf{inc}_y; \mathsf{inc}_y$	$\mathbb{N} \times \mathcal{B}$	0.309s
$(\mathsf{flip}\ x + \mathsf{flip}\ y + \mathsf{flip}\ z)^* \equiv (\mathsf{flip}\ x + \mathsf{flip}\ y + \mathsf{flip}\ z)^*$	$\mathbb{B}$	>30s (timeout)

Figure 6. Implementation microbenchmarks. We timeout at 30s because waiting longer is unreasonable for a prototyping tool.

trace refinement [3]. Our tracing semantics may be particularly well adapted for them, though they could generate KAT equations that fall outside of KMT's capabilities.

Smolka et al. [52] find an almost linear algorithm for checking equivalence of *guarded* KAT terms ($O(n \cdot \alpha(n))$, where α is the inverse Ackermann function), i.e., terms which use if and while instead of + and *, respectively. Their guarded KAT is completely abstract (i.e., actions are purely symbolic), while our KMTs are completely concrete (i.e., actions affect a clearly defined notion of state).

Our work is loosely related to Satisfiability Modulo Theories (SMT) [19]. Both aim to create an extensible framework where custom theories can be combined [41] and used to increase the expressiveness and power [53] of the underlying technique (SAT vs. KA). Some of our KMT theories implement satisfiability checking by calling out to Z3 [18].

The pushback requirement detailed in this paper is closely related to the classical notion of weakest precondition [6, 20, 47]. The pushback operation isn't quite a generalization of weakest preconditions because the various PB relations can change the program itself. 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 [24, 25, 27, 30, 42, 49], the problem remains undecidable in most cases; however, the pushback restrictions of "growth" of terms makes it possible for us to automatically lift the weakest precondition generation to loops in KAT. In fact, this is exactly what the normalization proof does when lifting tests out of the Kleene star operator.

The core technique we discuss here was first developed in Beckett et al.'s work on Temporal NetKAT [8]. Our work significantly extends that work: our normalization proof is explicit, rather than implicit; we separate proofs of correctness and termination of normalization; our treatment of negation is improved; we prove a new KAT theorem (Pushback-Neg); KMT is a general *framework* for proving completeness, while the Temporal NetKAT development is specialized to a particular instance; and Temporal NetKAT proof achieves limited completeness because of its limited understanding of LTL_f; we are able to achieve a more general result [9, 10]. Beckett et al. handles compilation to forwarding decision diagrams [51], while our presentation doesn't discuss compilation.

7 Conclusion

Kleene algebra modulo theories (KMT) is a new framework for extending Kleene algebra with tests with the addition of actions and predicates in a custom domain. KMT uses an operation that pushes tests back through actions to go from a decidable client theory to a domain-specific KMT. Derived KMTs are sound and complete with respect to a tracing semantics; we derive a decision procedure in an implementation that mirrors our formalism. The KMT framework captures common use cases and can reproduce *by mere composition* several results from the literature as well as several new results: we offer theories for bitvectors [28], natural numbers, unbounded sets, networks [1], and temporal logic [8]. Our ability to reason about unbounded state is novel. Our decision procedure follows our proof; automata-theoretic/coinductive approaches would be more efficient. Our approach isn't inherently limited to tracing semantics, as alternative regular interpretations could merge actions (as in KAT+B!, NetKAT, and Kleene algebra with equations [1, 28, 35]); future work could develop a relational semantics.

Acknowledgments

Dave Walker and Aarti Gupta provided valuable advice. Ryan Beckett was supported by NSF CNS award 1703493. The first two authors did preliminary work at Princeton University. The first and last authors did later work at Pomona College. Justin Hsu and Eric Koskinen provided advice and encouragement. Colin Gordon helped shepherd the paper; we thank the PLDI reviewers for helping improve the paper.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT:

Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 113–126.

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

[3] Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. 2019. Specification and inference of trace refinement relations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 178:1–178:30. https://doi.org/10.1145/3360604

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

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

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

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

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

[9] Eric Campbell and Michael Greenberg. 2021. Injecting Finiteness to Prove Completeness for Finite Linear Temporal Logic. *CoRR* abs/2107.06045 (2021). arXiv:2107.06045 https://arxiv.org/abs/2107.06045

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

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

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

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

[14] Ernie Cohen and Dexter Kozen. 2000. A note on the complexity of propositional Hoare logic. *ACM Trans. Comput. Log.* 1, 1 (2000), 171–174. https://doi.org/10.1145/343369.343404

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

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

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

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

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

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

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

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

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

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

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

[26] Murdoch J. Gabbay and Vincenzo Ciancia. 2011. Freshness and Name-restriction in Sets of Traces with Names. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbr&#252;cken, Germany) *(FOSSACS'11/ETAPS'11)*. Berlin, Heidelberg, 365–380.

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

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

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

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

[31] Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[52] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2020. Guarded Kleene algebra with tests: verification of uninterpreted programs in nearly linear time. *Proc. ACM Program. Lang.* 4, POPL (2020), 61:1–61:28. https://doi.org/10.1145/3371129

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