Kleene Algebra Modulo Theories

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

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

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

1 INTRODUCTION

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

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

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

Authors’ addresses: Ryan Beckett, Princeton University, rbeckett@cs.princeton.edu; Eric Campbell, Pomona College,
ehc02013@mymail.pomona.edu; Michael Greenberg, Pomona College, michael@cs.pomona.edu.
a general framework for automatically deriving sound, complete, and decidable KATs for client theories. The proof obligations of our approach are relatively mild and our approach is *compositional*: a client can compose smaller theories to form larger, more interesting KATs than might be tractable by hand. In addition to the equivalence decision procedure that comes from our completeness proof’s normalization routine, our theoretical framework has an automata theory that we prove correct. Our OCaml implementation allows users to compose a KAT with both decision procedures from small theory specifications. The automata are not only for verification, of course, they are useful for a variety of tasks such as compiling KATs to different implementations [8, 51]. We offer a fast path to a “minimum viable model” for those wishing to explore KATs formally or in code.

1.1 What is a KAT?

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

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

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

\[
\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg \beta \cdot \neg \gamma \equiv 0
\]

or that the original loop is equivalent to its unfolding:

\[
\alpha \cdot (\beta \cdot p \cdot q)^* \cdot \neg \beta \cdot \gamma \equiv \alpha \cdot (1 + \beta \cdot p \cdot q) \cdot (\beta \cdot p \cdot q \cdot \beta \cdot p \cdot q)^* \cdot \neg \beta \cdot \gamma
\]

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

As an example of such a domain-specific KAT, NetKAT showed how packet forwarding in computer networks can be modeled as simple While programs. Devices in a network must drop or permit packets (tests), update packets by modifying their fields (actions), and iteratively pass packets to and from other devices (loops). NetKAT extends KAT with two actions and one predicate: an action to write to packet fields, \( f \leftarrow v \), where we write value \( v \) to field \( f \) of the current packet; an action \( \text{dup} \), 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:

\[
dstIP \leftarrow 192.168.0.1 \cdot \text{dstPort} \leftarrow 4747 \cdot \text{dup}
\]

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

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

1.2 Using our framework: obligations for client theories

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

Our framework hinges on an operation relating predicates and operations called pushback, first used to prove relative completeness for Temporal NetKAT [8]. Pushback is a generalization of weakest preconditions. Given a primitive action \( \alpha \) and a primitive predicate \( \beta \), the client theory must be able to compute weakest preconditions, telling us how to go from \( \alpha \cdot \beta \) to some set of terms:

\[
\sum_{(i=0)}^{n} \alpha_i \cdot \beta = \alpha_0 \cdot \beta + \alpha_1 \cdot \beta + \ldots
\]

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. Given the client’s notion of weakest preconditions, we can alter programs to take an arbitrary term and normalize it into a form where
all of the predicates appear only at the front of the term, a convenient representation both for our completeness proof (Sec. 2.4) and our automata-theoretic implementation (Secs. 4 and 5).

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

As an example, in NetKAT, for different fields \( f \) and \( f' \), we can use the network axioms to derive the equivalence: \( (f \leftarrow v \cdot f' = v') \equiv (f' = v' \cdot f \leftarrow v) \), which satisfies the pushback requirements.

For Temporal NetKAT, which adds rich temporal predicates such as \( \diamond \circ (\text{dstPort} = 4747) \) (the destination port was 4747 at some point before the previous state), we can use the domain axioms to prove the equivalence \( (f \leftarrow v \cdot \diamond \circ a) \equiv (\diamond \circ a + a) \cdot f \leftarrow v \), which also satisfies the pushback requirements of equivalence and non-increasing measure.

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

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

### 1.3 Example: incrementing naturals

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

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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>( x &gt; n )</td>
</tr>
<tr>
<td>( \pi )</td>
<td>( \text{inc}_x \mid x := n )</td>
</tr>
<tr>
<td>( \text{sub}(x &gt; n) )</td>
<td>( { x &gt; m \mid m \leq n } )</td>
</tr>
</tbody>
</table>

### Semantics

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n \in \mathbb{N} )</td>
<td>( x \in \mathcal{V} )</td>
</tr>
<tr>
<td>( \text{State} )</td>
<td>( \mathcal{V} \to \mathbb{N} )</td>
</tr>
<tr>
<td>( \text{pred}(x &gt; n, t) )</td>
<td>( \text{last}(t)(x) &gt; n )</td>
</tr>
<tr>
<td>( \text{act}(\text{inc}_x, \sigma) )</td>
<td>( \sigma[x \mapsto \sigma(x) + 1] )</td>
</tr>
<tr>
<td>( \text{act}(x := n, \sigma) )</td>
<td>( \sigma[x \mapsto n] )</td>
</tr>
</tbody>
</table>

### Axioms

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

### Weakest precondition

\[
\begin{align*}
\text{inc}_y \cdot (x > n) \cdot \text{WP}(x > n) & \\
\text{inc}_x \cdot (x > n) \cdot \text{WP}(x > n - 1) & \quad \text{when } n \neq 0 \\
\text{inc}_x \cdot (x > 0) \cdot \text{WP}1 &
\end{align*}
\]

### Implementation

**Fig. 1. IncNat, increasing naturals**

(inadmissible!) term \( x := 0 ; y := 0 ; (\text{inc}_x)^* \cdot (\text{inc}_y)^* \cdot x = y \) describes programs with balanced increments of \( x \) and \( y \). For the same reason, we cannot safely add a decrement operation \( \text{dec}_x \).

Either of these would allow us to define counter machines, leading inevitably to undecidability.

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

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

- **Type** \( a = \text{Get of string} \cdot \text{int} \) (* alpha ::= x > n *)
- **Type** \( p = \text{Increment of string} \) (* pi ::= inc x *)

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

Publication date: July 2018.
\begin{verbatim}
| (Increment x, Gt (y, j)) when x = y ->
PSet.singleton ~ cmp K.Test.compare (K.theory (Gt (y, j - 1)))
| (Assign (x, i), Gt (y, j)) when x = y ->
PSet.singleton ~ cmp K.Test.compare (if i > j then K.one () else K.zero())
| _ -> PSet.singleton ~ cmp K.Test.compare (K.theory a)
let rec subterms x =
    match x with
    | Gt (_, 0) -> PSet.singleton ~ cmp K.Test.compare (K.theory x)
    | Gt (v, i) -> PSet.add (K.theory x) (subterms (Gt (v, i - 1)))

(* decision procedure for the predicate theory *)
let satisfiable (a: K.Test.t) = ...
end
\end{verbatim}

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

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

1.4 Contributions

We claim the following contributions:

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

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

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

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

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

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

2.1 Semantics

The first step in turning the client theory $T$ into a KAT is to define a semantics (Fig. 3). We can give any KAT a trace semantics: the meaning of a term is a trace $t$, which is a non-empty list of log entries $l$. Each log entry records a state $\sigma$ and (in all but the initial state) a primitive action $\pi$. 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 $T^*$ term on a state $\sigma$, 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 $\sigma_i = \mathit{act}(\pi_i, \sigma_{i-1})$ 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 $\alpha$ and a trace — predicates can examine the entire trace — returning true or false. When the pred function returns true, we return the singleton set holding our input trace; when pred returns false, we return the empty set. (Composite predicates follow this same pattern, always returning either a singleton set holding their input trace or the empty set.) It’s acceptable for the pred function to recursively call the denotational
Proving that the equational theory is sound is relatively straightforward: we only depend on the client’s act and pred functions, and none of our KAT axioms (Fig. 3) even mention the client’s primitives. We believe the pushback negation theorem (Pushback-Neg) is novel (though it holds in all KATs). Our soundness proof naturally enough requires that any equations the client theory adds need to be sound in our trace semantics. We do need to use several KAT theorems in our completeness proof (Fig. 3, Consequences), the most complex being star expansion (Star-Expand), which we take from Temporal NetKAT [8]; we believe Pushback-Neg is a novel theorem that holds in all KATs.

2.2 Soundness

Fig. 3. Semantics and equational theory for $T^*$

The client’s act function takes a primitive action $\pi$ 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.

Kleene Algebra axioms

<table>
<thead>
<tr>
<th>Rule</th>
<th>Axiom</th>
<th>Short Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \cdot 1 \equiv t$</td>
<td>$K\text{-Seq-Zero}$</td>
<td></td>
</tr>
<tr>
<td>$p \cdot r \equiv r \cdot p$</td>
<td>$K\text{-Seq-Assoc}$</td>
<td></td>
</tr>
<tr>
<td>$p \cdot (q + r) \equiv (p \cdot q) + (p \cdot r)$</td>
<td>$K\text{-Seq-Idem}$</td>
<td></td>
</tr>
<tr>
<td>$p \equiv p \cdot 1$</td>
<td>$K\text{-Seq-One}$</td>
<td></td>
</tr>
<tr>
<td>$p \equiv 0 \equiv 0$</td>
<td>$K\text{-Seq-0}$</td>
<td></td>
</tr>
<tr>
<td>$p \equiv 1$</td>
<td>$K\text{-Seq-1}$</td>
<td></td>
</tr>
<tr>
<td>$p + (q + r) \equiv (p + q) + r$</td>
<td>$K\text{-Plus-Assoc}$</td>
<td></td>
</tr>
<tr>
<td>$p + q \equiv q + p$</td>
<td>$K\text{-Plus-Comm}$</td>
<td></td>
</tr>
<tr>
<td>$p + 0 \equiv p$</td>
<td>$K\text{-Plus-Zero}$</td>
<td></td>
</tr>
<tr>
<td>$p + p \equiv p$</td>
<td>$K\text{-Plus-Idem}$</td>
<td></td>
</tr>
<tr>
<td>$p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$</td>
<td>$K\text{-Plus-Dist}$</td>
<td></td>
</tr>
</tbody>
</table>

Boolean Algebra axioms

<table>
<thead>
<tr>
<th>Rule</th>
<th>Axiom</th>
<th>Short Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + (b \cdot c) \equiv (a + b) \cdot (a + c)$</td>
<td>$B\text{-Plus-Dist}$</td>
<td></td>
</tr>
<tr>
<td>$a + 1 \equiv 1$</td>
<td>$B\text{-Plus-One}$</td>
<td></td>
</tr>
<tr>
<td>$a + \neg a \equiv 1$</td>
<td>$B\text{-Plus-One}$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot b \equiv b \cdot a$</td>
<td>$B\text{-Plus-One}$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot \neg a \equiv 0$</td>
<td>$B\text{-Plus-One}$</td>
<td></td>
</tr>
<tr>
<td>$a \equiv a$</td>
<td>$B\text{-Plus-One}$</td>
<td></td>
</tr>
<tr>
<td>$p \cdot a \equiv b \cdot p \rightarrow p \cdot \neg a \equiv \neg b \cdot p$</td>
<td>Pushback-Neg</td>
<td></td>
</tr>
<tr>
<td>$p \cdot (q \cdot r)^* \equiv (p \cdot q)^* \cdot p$</td>
<td>Sliding</td>
<td></td>
</tr>
<tr>
<td>$(p + q)^* \equiv p^* \cdot (q \cdot p)^*$</td>
<td>Denesting</td>
<td></td>
</tr>
<tr>
<td>$p \cdot a \equiv a \cdot q + r$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p^* \cdot a \equiv (a + p^* \cdot r) \cdot q^*$</td>
<td>Star-Inv</td>
<td></td>
</tr>
<tr>
<td>$p \cdot a \equiv a \cdot q + r$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p \cdot a \equiv (a \cdot q + r) \cdot (q + r)^*$</td>
<td>Star-Expand</td>
<td></td>
</tr>
</tbody>
</table>

Consequences

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

\[
\begin{align*}
p \leq q & \iff p + q \equiv q \\
\end{align*}
\]
**Theorem 2.1 (Soundness of $\mathcal{T}^*$).** If $\mathcal{T}$’s equational reasoning is sound ($p \equiv_T q \Rightarrow \llbracket p \rrbracket = \llbracket q \rrbracket$) then $\mathcal{T}^*$’s equational reasoning is sound ($p \equiv q \Rightarrow \llbracket p \rrbracket = \llbracket q \rrbracket$).

**Proof.** By induction on the derivation of $p \equiv q$.\hfill $\square$

### 2.3 Normalization via pushback

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

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

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

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

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

Let the set of restricted actions $\mathcal{T}_{RA}$ be the subset of $\mathcal{T}^*$ where the only test is $1$. We will use metavariables $m$, $n$, and $l$ to denote elements of $\mathcal{T}_{RA}$. Let the set of normal forms $\mathcal{T}_{nf}^*$ be a set of pairs of tests $a_i \in \mathcal{T}_{pred}^*$ and restricted actions $m_i \in \mathcal{T}_{RA}$. We will use metavariables $t$, $u$, $v$, $w$, $x$, $y$, and $z$ to denote elements of $\mathcal{T}_{nf}^*$; 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, $\mathcal{T}_{nf}^*$, 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 $\mathcal{T}_{nf}^*$.

Our normalization algorithm uses the maximal subterm ordering as its termination measure. Due to space constraints, we provide the formal definitions of maximal tests and subterms in the supplemental material. Here we simply give intuition for the two relevant high-level operations: $\text{mt}(x) \subseteq \mathcal{T}^*_{pred}$ computes the maximal tests of a normal form $x$, which are those tests that are not subterms of any other test; the maximal subterm ordering $x \preceq y$ for normal forms holds when the subterms of $x$’s maximal tests are a subset of the subterms of $y$’s maximal tests. Informally, we have $x \preceq y$ when every test in $x$ is somehow “covered” by a test in $y$; we have $x < y$ when $x \preceq y$ and $y$ has some maximal test $x$ that does not.

Our definition of subterms needs the client theory to identify the subterms of its primitives via a function $\text{sub}_{\mathcal{T}}$ such that (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.

**Lemma 2.2 (Splitting).** If $a \in \text{mt}(x)$, then there exist $y$ and $z$ such that $x \equiv a \cdot y + z$ and $y < x$ and $z < x$.

1Full proofs with all necessary lemmas are available in an extended version of this paper in the supplementary material.
Splitting is the key lemma for making progress pushing tests back, allowing us to take a normal form and slowly push its maximal tests to the front; its proof follows from a chain of lemmas given in the supplementary material.

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

**Definition 2.3 (Weakest preconditions).** The weakest precondition operation of the client theory is a relation \( WP \subseteq T_\alpha \times T_\alpha \times \mathcal{P}(\mathcal{T}_\text{prec}) \), where \( T_\alpha \) are the primitive actions and \( \mathcal{T}_\alpha \) are the primitive predicates of \( \mathcal{T} \). We write the relation as \( \pi \cdot \alpha WP \sum a_i \cdot \pi \) and read it as “\( \alpha \) pushes back through \( \pi \) to yield \( \sum a_i \cdot \pi \)” ; the second \( \pi \) is redundant but written for clarity. 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. 4). The client’s \( WP \) relation need not be a function, nor do the \( a_i \) need to be obviously related to \( \alpha \) or \( \pi \) in any way. Even when the \( WP \) relation is a function, the \( PB \) relation will generally not be a function. While \( WP \) computes the classical weakest precondition, the \( PB \) relations do something different: when pushing back we have the freedom to change the program itself—not normally an option for weakest preconditions (see Sec. 4).

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

For sequences, we can recursively take \( p \cdot q \) and normalize \( p \) into \( x = \sum_i a_i \cdot m_i \) and \( q \) into \( y = \sum j \cdot n_j \). 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 (\( \text{Join} \)); we write \( x \cdot y PB^1 z \) to mean that the concatenation of \( x \) and \( y \) is equivalent to the normal form \( z \)—the \( \cdot \) is suggestive notation.

For Kleene star, we can take \( p^* \) and normalize \( p \) into \( x = \sum_i a_i \cdot m_i \), but \( x^* \) isn’t a normal form—we need to somehow move all of the tests out of the star and to the front. We do so with the \( PB^* \) relation (Fig. 4), writing \( x^* PB^* y \) 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 \( PB^1 \). There are four possible ways to treat \( x \), based on how it splits (Lemma 2.2): if \( x = 0 \), then our work is trivial since \( 0^* \equiv 1 \) (\( \text{StarZero} \)); 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 \( x \) (\( \text{Slide} \)) or by using the KAT expansion lemma (\( \text{Expand} \)); if \( x \) splits into \( a \cdot x' + z \), we use the KAT denesting lemma to pull \( a \) out before recurring on what remains (\( \text{Denest} \)).

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

We show that our notion of pushback is correct in two steps. First we prove that pushback is partially correct, i.e., if we can form a derivation in the pushback relations, the right-hand sides are equivalent to the left-hand-sides (Theorem 2.4). Once we’ve established that our pushback
Fig. 4. Normalization for $T^*$
relations’ derivations mean what we want, we have to show that we can find such derivations; here we use our maximal subterm measure to show that the recursive tangle of our PB relations always terminates (Theorem 2.5).

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

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

**Theorem 2.5 (Pushback existence).** For each of the PB relations, every left side relates to a right side that is no larger, e.g., for all \( x \) there exists \( y \) \( \preceq x \) 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 2.2) to show that derivations exist followed by subterm ordering congruence to find orderings to apply the IH.

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

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

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

**Proof.** By induction on \( p \), using Theorems 2.5 and 2.4 in the SEQ and STAR case.

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

### 2.4 Completeness

We prove completeness—if \( \llbracket p \rrbracket = \llbracket q \rrbracket \) then \( p \equiv q \)—by normalizing \( p \) and \( q \) and comparing the resulting terms. Our completeness proof uses the completeness of Kleene algebra (KA) as its foundation: the set of possible traces of actions performed for a restricted (test-free) action in our denotational semantics is a regular language, and so the KA axioms are sound and complete for it. In order to relate our denotational semantics to regular languages, we define the regular interpretation of restricted actions \( m \in T^*_{RA} \) in the conventional way and then relate our denotational semantics to the regular interpretation (Fig. 5). Readers familiar with NetKAT’s completeness proof may notice that we’ve omitted the language model and gone straight to the regular interpretation. We’re able to shorten our proof because our tracing semantics is more directly relatable to its regular interpretation, and because our completeness proof separately defers to the client theory’s decision procedure for the predicates at the front. Our normalization routine—the essence of our proof—only uses the KAT axioms and doesn’t rely on any property of our tracing semantics. We conjecture that one could prove a similar completeness result and derive a similar decision procedure with a merging, non-tracing semantics, like in NetKAT or KAT+B! [1, 29]. We haven’t attempted the proof or an implementation.
Lemma 2.7 (Labels are regular). \{\text{label}([m]|(\sigma, \bot)) | \sigma \in \text{State}\} = \mathcal{R}(m)

Proof. By induction on the restricted action \( m \).

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

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

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

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

\[
\hat{x} = a_1 \cdot a_2 \cdots a_n \cdot m_1 \cdot m_2 \cdots m_n \\
+ \neg a_1 \cdot a_2 \cdots a_n \cdot m_2 \cdots m_n \\
+ a_1 \cdot \neg a_2 \cdots a_n \cdot m_1 \cdots m_n \\
+ \ldots \\
+ \neg a_1 \cdot \neg a_2 \cdots a_n \cdot m_n
\]

and similarly for \( \hat{y} \). We can find \( x \equiv \hat{x} \) via distributivity (BA-PLUS-DIST) and the excluded middle (BA-EXCL-MID).

Given normal forms with locally disjoint cases, we can take the Cartesian product of \( \hat{x} \) and \( \hat{y} \), which allows us to do a syntactic comparison on each of the predicates. Let \( \tilde{x} \) and \( \tilde{y} \) be the extension of \( \hat{x} \) and \( \hat{y} \) with the tests from the other form, giving us \( \tilde{x} = \sum i,j c_i \cdot d_j \cdot l_i \) and \( \tilde{y} = \sum i,j c_i \cdot d_j \cdot m_j \).

Extending the normal forms to be disjoint between the two normal forms is still provably equivalent using commutativity (BA-SEQ-COMM), distributivity (BA-PLUS-DIST), and the excluded middle (BA-EXCL-MID).

Now that each of the predicates are syntactically uniform and disjoint, we can proceed to compare the commands. But there is one final risk: what if the \( c_i \cdot d_j \equiv 0' \)? Then \( l_i \) and \( a_j \) could safely be different. We therefore use the client’s emptiness checker to eliminate those cases where the expanded tests at the front of \( \tilde{x} \) and \( \tilde{y} \) are equivalent to zero, which is sound by the client theory’s completeness and zero-cancellation (KA-ZERO-SEQ and KA-SEQ-ZERO).
Finally, we can defer to deductive completeness for KA to find proofs that the commands are equal. To use KA’s completeness to get a proof over commands, we have to show that if our commands have equal denotations in our semantics, then they will also have equal denotations in the KA semantics. We’ve done exactly this by showing that restricted actions have regular interpretations: because the zero-canceled \( x \) and \( y \) are provably equal, soundness guarantees that their denotations are equal. Since their tests are pairwise disjoint, if their denotations are equal, it must be that any non-canceled commands are equal, which means that each label of these commands must be equal—and so \( \mathcal{R}(l_i) = \mathcal{R}(o_j) \) (Lemma 2.7). By the deductive completeness of KA, we know that KA \( \vdash l_i \equiv o_j \). Since we have the KA axioms in our system, then \( l_i \equiv o_j \); by reflexivity, we know that \( c_i \cdot d_j \equiv c_i \cdot d_j \), and we have proved that \( x \equiv y \). By transitivity, we can see that \( x \equiv y \) and so \( x \equiv y \) and \( p \equiv q \), as desired.

3 CASE STUDIES

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

3.1 Bit vectors

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

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

3.2 Disjoint products

Given two client theories, we can combine them into a disjoint product theory, \( \text{Prod}(\mathcal{T}_1, \mathcal{T}_2) \), where the states are products of the two sub-theory’s states and the predicates and actions from \( \mathcal{T}_1 \) can’t...
We define a KMT for unbounded sets parameterized on a theory of expressions $E$. We allowed expressions in all positions, variables valued as either booleans or (increasing) naturals; the product theory lets us directly project the trace state to the $i$th component. Our state has two parts: a set data type supports just one operation: $\text{add}(x, e)$ adds the value of expression $e$ to set $x$ (we could add $\text{del}(x, e)$, but we omit it to save space). It also supports a single test: $\text{in}(x, c)$ checks if the constant $c$ is contained in set $x$. The idea is that $e \in E$ refers to expressions with, say, variables $x$ and constants $c$. We allow arbitrary expressions $e$ in some positions and constants $c$ in others. (If we allowed expressions in all positions, WP wouldn’t necessarily be non-increasing.)

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

### Syntax

$$
\begin{align*}
\alpha & ::= \alpha_1 | \alpha_2 \\
\pi & ::= \pi_1 | \pi_2 \\
\text{sub}(\alpha_i) & = \text{sub}(\pi_i)
\end{align*}
$$

### Semantics

$$
\begin{align*}
\text{State} & = \text{State}_1 \times \text{State}_2 \\
\text{pred}(\alpha_i, t) & = \text{pred}_1(\alpha_i, t) \\
\text{act}(\pi_i, \sigma) & = \sigma[\sigma_1 \mapsto \text{act}_1(\pi_i, \sigma_1)]
\end{align*}
$$

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

$$
\begin{align*}
\pi_1 \cdot \alpha_2 & \equiv \alpha_2 \cdot \pi_1 & \text{L-R-Comm} \\
\pi_2 \cdot \alpha_1 & \equiv \alpha_1 \cdot \pi_2 & \text{R-L-Comm}
\end{align*}
$$

### Syntax

$$
\begin{align*}
\alpha & ::= \text{in}(x, c) | e = c | \alpha e \\
\pi & ::= \text{add}(x, e) | \pi e \\
\text{sub}(\text{in}(x, c)) & = \{ \text{in}(x, c) \} \cup \text{sub}(-(e = c)) \\
\text{sub}(e = c) & = \text{sub}(e = c) \\
\text{sub}(\alpha e) & = \text{sub}(\alpha e)
\end{align*}
$$

### Semantics

$$
\begin{align*}
c & \in C \\
e & \in E \\
x & \in \mathcal{V} \\
\text{State} & = (\mathcal{V} \rightarrow \mathcal{P}(C)) \times (E \rightarrow C) \\
\text{pred}(\text{in}(x, c), t) & = \text{last}(t_2)(c) \in \text{last}(t_1)(x) \\
\text{pred}(\alpha e, t) & = \text{pred}(\alpha e, t_2) \\
\text{act}(\text{add}(x, e), \sigma) & = \sigma[\sigma_1[x \mapsto \sigma_1(x) \cup \{\sigma(e)\}]] \\
\text{act}(\pi e, \sigma) & = \sigma[\sigma_2 \mapsto \text{act}(\pi e, \sigma_2)]
\end{align*}
$$

### Axioms extending $E$

$$
\begin{align*}
\text{add}(y, e) \cdot \text{in}(x, c) & \equiv \text{in}(x, c) \cdot \text{add}(y, e) & \text{Add-Comm} \\
\text{add}(x, e) \cdot \text{in}(x, c) & \equiv (e = c) + \text{in}(x, c) \cdot \text{add}(x, e) & \text{Add-In} \\
\text{add}(x, e) \cdot \alpha e & \equiv \alpha e \cdot \text{add}(x, e) & \text{Add-Comm2}
\end{align*}
$$

### Syntax

$$
\begin{align*}
\text{add}(y, e) \cdot \text{in}(x, c) & \equiv \text{in}(x, c) \cdot \text{add}(y, e) \\
\text{add}(x, e) \cdot \text{in}(x, c) & \equiv ((e = c) + \text{in}(x, c)) \cdot \text{add}(x, e) \\
\text{add}(x, e) \cdot \alpha e & \equiv \alpha e \cdot \text{add}(x, e)
\end{align*}
$$

### Axioms extending $\mathcal{E}$

To instantiate the Set theory, we need a few things: expressions $E$, a subset of constants $C \subseteq E$, and predicates for testing (in)equality between expressions and constants ($e = c$ and $e \neq c$). We can not, in general, expect tests for equality of non-constant expressions, as it may cause us to accidentally define a counter machine.) We treat these two extra predicates as inputs, and expect that they have well behaved subterms. Our state has two parts: $\alpha_1 : \mathcal{V} \rightarrow \mathcal{P}(C)$ records the current sets for each set in $\mathcal{V}$, while $\alpha_2 : E \rightarrow C$ evaluates expressions in each state. Since each state has its own evaluation function, the expression language can have actions that update $\alpha_2$.

### 3.3 Unbounded sets

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

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

Fig. 8. Set($E$), unbounded sets over expressions
The last and since predicates together are enough to encode the rest of LTL.

3.4 Past-time linear temporal logic

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

LTL\(_f\) adds just two predicates: \(\Box a\), pronounced “last \(a\)”, means \(a\) held in the prior state; and \(a \diamond b\), 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 \(\Box a\) is false at the beginning (what can be true in a state that never happened?), and \(a \diamond b\) degenerates to \(b\) at the beginning of time. The last and since predicates together are enough to encode the rest of LTL\(_f\); encodings are given below the syntax. Weakest preconditions uses inference rules: to push back \(S\), we unroll \(a \diamond (a \diamond b) + b\); pushing last through an action is easy, but pushing back \(a\) or \(b\) recursively uses the PB\(_a\) judgment. Adding these rules leaves our judgments monotonic, and if \(\pi \cdot a\) PB\(_a\) \(x\), then \(x = \sum a \cdot \pi\). In this case, our implementation’s recursive modules are critical—they allow us to use the derived pushback inside our definition of weakest preconditions.

The equivalence axioms come from Temporal NetKAT [8]; the deductive completeness result for these axioms comes from Campbell and Greenberg’s work, which proves deductive completeness.
for an axiomatic framing and then relates those axioms to our equations [10, 11]; we could have also used Rosu’s proof with coinductive axioms [45].

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

$$j := j + 2 \cdot \Box (j \leq 200) \equiv j := j + 2 \cdot (j \leq 200 \cdot \Box (j \leq 200))$$

Pushing the temporal test back through the action reveals that $j$ is never greater than 200 if before the action $j$ was not greater than 198 in the previous state and $j$ 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 $\text{LTL}_f$: we can generate a complete equational theory for $\text{LTL}_f$ 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 $r \equiv r \cdot \Box (j \leq 200)$. 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 $r \cdot \Box (j \leq 200)$ is empty—if so, there are no satisfying traces.

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

### 3.5 Tracing NetKAT

We define NetKAT as a KMT over packets, which we model as functions from packet fields to values (Fig. 10). KMT’s trace semantics diverge slightly from NetKAT’s: like KAT+B! (Sec. 3.1; [29]), NetKAT normally merges adjacent writes. If the policy analysis demands reasoning about the history of packets traversing the network—reasoning, for example, about which routes packets actually take—the programmer must insert dups to record relevant moments in time. From our
perspective, NetKAT very nearly has a tracing semantics, but the traces are selective. If we put an implicit dup before every field update, NetKAT has our tracing semantics.

### 3.6 Temporal NetKAT

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

### 4 AUTOMATA

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

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

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

---

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

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

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

4.1 Constructing KMT automata

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

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

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

\[ \begin{array}{ll}
\text{Derivative} & D : \mathcal{T}_\ell^* \rightarrow \mathcal{P}(\mathcal{T}_\ell^* \times \mathcal{T}_\ell^* \times \mathcal{T}_\ell^*) \\
D(0) & = 0 \\
D(1) & = 0 \\
D(\alpha) & = 0 \\
D(\pi^\ell) & = \{ (1, \pi^\ell, 1) \} \\
D(p + q) & = D(p) \cup D(q) \\
D(p \cdot q) & = D(p) \circ q \cup E(p) \circ D(q) \\
D(p^*) & = D(p) \circ p^*
\end{array} \]

\[ E(0) = 0 \]
\[ E(1) = 1 \]
\[ E(\alpha) = \alpha \]
\[ E(\pi^\ell) = 0 \]
\[ E(p + q) = E(p) + E(q) \]
\[ E(p \cdot q) = E(p) \cdot E(q) \]
\[ E(p^*) = 1 \]

\[ \mathcal{D}(p) \circ q = \{ (d, \pi^\ell, k \cdot q) \mid (d, \pi^\ell, k) \in \mathcal{D}(p) \} \quad q \circ \mathcal{D}(p) = \{ (q \cdot d, \pi^\ell, k) \mid (d, \pi^\ell, k) \in \mathcal{D}(p) \} \]

\[ A_\pi(p) = (S, s_0, \epsilon, \delta) \]
\[ S = \{ 0 \} \cup \text{labels}(p) \]
\[ s_0 = 0 \]
\[ \epsilon \; t \iff t \in \mathcal{E}(k) \mathcal{E}(t) \]
\[ \delta \; t = \{ (\sigma^\ell, \pi^k) \mid (d, \pi^\ell, k) \in \mathcal{D}(k) \land t \in \mathcal{D}(k) \land \mathcal{E}(\pi^k) \mathcal{E}(t) \} \]

\[ \begin{array}{ll}
\mathcal{A}_\pi(p), \ell \text{ accepts } t ; \cdot & \iff \epsilon \; t \\
\mathcal{A}_\pi(p), \ell \text{ accepts } t ; (\sigma, \pi^k) t' & \iff (\sigma, \pi^k) \in \delta \; t \land \mathcal{A}_\pi(p), \ell \text{ accepts } t(\sigma, \pi^k); t'
\end{array} \]

Fig. 12. KMT partial derivatives and automata

Term automaton trace acceptance

\[ \text{accepts } \subseteq \text{Automaton } \times \text{States } \times \text{Trace } \times (\text{State } \times \mathcal{T}_\pi)^* \]

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

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

**Lemma 4.1 (Derivative correct).** For all programs $p$ where each primitive action $\pi$ is augmented with a unique label $\ell$,

1. $p = E(p) + \sum_{(d, \pi^\ell, k) \in \mathcal{D}(p)} d \cdot \pi^\ell \cdot k$, and
2. For all labels $\ell$ in $p$, there exist unique $d$ and $k$ such that $(d, \pi^\ell, k) \in \mathcal{D}(p)$.

**Proof.** For (1), we go by induction on $p$, using DENESTING in the star case. For (2), let $\pi$ and $\ell$ be given; we go by induction on $p$. □

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

**Proof.** By induction on the length of $t'$, leaving $t$ general. □
\( A_\alpha(a) = (S, s_0, \epsilon, \delta) \)  
Theory automaton

\( S = 2^{\text{sub}(a)} \)  
States

\( s_0(t) = \{ b \in \text{sub}(a) \mid t \in [b](t) \} \)  
Initial state selector

\( \text{serialize}(A) = \Pi_{a \in A} a \)  
Serialization of predicate sets

\( \epsilon \in T \)  
Acceptance condition

\( \delta \in T \)  
Transition relation

\begin{equation*}
\text{traces} \subseteq \text{Automaton}_\alpha \times S \times \text{Trace} \times (\text{State} \times T_\alpha)^\ast
\end{equation*}

Fig. 13. Theory automata

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

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

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

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

Continuing our example (Fig. 11 (middle)), there is a transition from state 2 to state 3 for action \( \text{inc}_x \). State 3 is labeled with \( \{ 1, x > 0, x > 1, x > 2, \diamond x > 2 \} \) and state 2 is labeled with \( \{ 1, x > 0, x > 1 \} \). We compute \( \text{inc}_x \cdot \diamond x > 2 \text{ WP} (\diamond x > 2 + x > 1) \). Therefore, \( \diamond x > 2 \) should be
\( \mathcal{A}_{\text{KMT}}(p) = (S, s_0, \epsilon, \delta) \)

\( S = S^{\mathcal{A}_a(p)} \times S^{\mathcal{A}_a(a_1)} \times \cdots \times S^{\mathcal{A}_a(a_n)} \) where \( a_i \in \mathcal{A}_\pi(p) \)

KMT automaton

States

\( s_0(t) = \lambda t. (s_0, s_0^{\mathcal{A}_a(a_1)}(t), \ldots, s_0^{\mathcal{A}_a(a_n)}) \)

Initial state selector

\( \epsilon \ s \ t \iff \epsilon^{\mathcal{A}_a(a_i)} \ s \ i \ t \) where \( \epsilon^{\mathcal{A}_a(p)} \ s \ t = a_i \)

Acceptance condition

\( \delta \ s \ t = \{(\sigma, \pi^{t'}, (t', \delta^{\mathcal{A}_a(a_i)} s.1 t, \ldots, \delta^{\mathcal{A}_a(a_n)} s.n t)) \} \)

transition condition

\( \langle a_i, \pi^{t'}, k \rangle \in D(k_{t'}) \land \epsilon^{\mathcal{A}_a(a_i)} s.1 t \land t(\sigma', \pi^{t'}) \in \|\pi^{t'}\|(t) \) \n
Transition relation

KMT automaton acceptance

\[ \text{accepts} \subseteq \text{Automaton}_{\text{KMT}} \times S \times \text{Trace} \times (\text{State} \times T_\pi)^* \]

\( \mathcal{A}_{\text{KMT}}(p), s \text{ traces } t; \bullet \iff \epsilon \ s \ t \)

Accepting state

\( \mathcal{A}_{\text{KMT}}(p), s \text{ traces } t; (\sigma, \pi)t' \iff (\sigma, \pi, s') \in \delta \ s \ t \land \mathcal{A}_{\text{KMT}}(p), s' \text{ accepts } t(\sigma, \pi); t' \)

Taking a step

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

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

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

Lemma 4.3 (Theory automaton correct). \( t \in \|\text{serialize}(A)\|(t) \iff \mathcal{A}_a(a), A \text{ traces } t; t' \)

Proof. By induction on the length of \( t' \), leaving \( t \) general.

\[ \square \]

4.2 KMT automata

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

For example, in Fig. 11, the quasi-intersected automata (right) replaces instances of the \( \Diamond x > 2 \) condition in state 0 of the term automaton, with the acceptance condition from the corresponding state in the theory automaton. In state \((2,0)\) this is true, while in states \((1,0)\) and \((0,0)\) this is false.

For transitions with the same action \( \pi \), the quasi-intersection takes the conjunction of each edge’s tests. Formally, we define the KMT automaton as a 4-tuple \( (S, s_0, \epsilon, \delta) \), where the states are those of \( \mathcal{A}_a(p) \) along with those of \( \mathcal{A}_a(a) \) for every predicate \( a \) that occurs in \( \mathcal{A}_\pi(p) \). The initial state selector \( s_0 \), acceptance condition \( \epsilon \), and transition relation \( \delta \) are all defined as composites of the term and theory automata, using the appropriate theory automaton to implement the transition relation \( \delta \) and acceptance condition \( \epsilon \).

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

\[ \text{Vol. 1, No. 1, Article } \]
Lemma 4.4 (KMT automaton correct).

\[
\begin{align*}
\text{tt}' &\in [\llbracket k \rrbracket(t) \text{ and } t \in \llbracket \text{serialize}(A_i) \rrbracket(t) \text{ iff } A_{\text{KMT}}(p), (\ell, A_1, \ldots, A_n) \text{ accepts } t; t'.
\end{align*}
\]

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

4.3 Equivalence checking using automata

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

5 IMPLEMENTATION

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

```ocaml
module P = Product(IncNat)(Boolean)
module A = Automata(P.K) (* automata-theoretic decision procedure *)
module D = Decide(P) (* normalization-based decision procedure *)
```

let a = P.K.parse "y<1; (a=F + a=T; inc(y)); y>0" in
let b = P.K.parse "y<1; a=T; inc(y)" in
assert (A.equivalent (A.of_term a) (A.of_term b));
assert (D.equivalent a b)

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

5.1 Optimizations

We’ve implemented smart constructors, which hash-cons and also automatically rewrite common identities (e.g., constructing \( p \cdot 1 \) will simply return \( p \); constructing \( (p^*)^* \) will simply return \( p^* \)). Client theories can extend the smart constructors to witness theory-specific identities. Client theories can implement custom solvers or rely on \( \text{Z3} \) embeddings—custom solvers are typically faster. These optimizations are partly responsible for the speed of our normalization routine (when it avoids the costly \( \text{Denest} \) case).
We haven’t particularly optimized our automata implementation. Two particular opportunities for optimization stand out, both of which focus on reducing the state space of the theory automata. First, most client-theory predicates only consider the most recent state, in which case we need not generate a theory automaton at all. Second, the formal presentation of theory automata generates one automaton per predicate, the states of which are subsets of subterms of that predicate—an exponential blowup. While convenient for the proof, many predicates will share subterms—so we pay the cost of blowup more than once, tracking the same subterms in more than one theory automaton. We could instead generate a single theory automaton, where a state is a set drawn from subterms of all of the predicates in the term automaton, which would reduce some of the state-space blowup.

6 EVALUATION

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

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

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

7 RELATED WORK

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

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

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

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

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

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

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

8 CONCLUSION

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

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

REFERENCES


