Kleene Algebra Modulo Theories

MICHAEL GREENBERG, Pomona College
RYAN BECKETT, Microsoft Research
ERIC CAMPBELL, Cornell University

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.

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. 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. 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.

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. 2020. Kleene Algebra Modulo Theories. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 47 pages.

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 [3, 7, 12–14, 41] 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, 22, 42, 44, 56].

Considering KAT's success in networks, we believe other domains would benefit from programming languages where program equivalence is decidable. However, extending a KAT for a particular domain remains a challenging task even for experts familiar with KATs and their metatheory. To build a custom KAT, experts must craft custom domain primitives, derive a collection of new

Authors' addresses: Michael Greenberg, Pomona College, michael@cs.pomona.edu; Ryan Beckett, Microsoft Research, Ryan.Beckett@microsoft.com; Eric Campbell, Cornell University, ehc86@cornell.edu.

0164-0925/2020/1-ART1
https://doi.org/

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, 60], after a long series of papers that resembled, but did not use, the KAT framework [21, 31, 45, 51]. 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 [15, 39, 46, 50]. 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, 37].

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 a general framework for automatically deriving sound, complete, and decidable KATs with tracing semantics for client theories. To demonstrate the effectiveness of our approach, we not only reproduce results from the literature (e.g., tracing variants of bit vectors and NetKAT), but we also derive new KATs that go behind the existing, finite-state KATs to KATs using monotonically increasing, unbounded 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 an equivalence decision procedure. Our OCaml implementation allows users to compose a KAT with both decision procedures from small theory specifications. We offer a fast path to a "minimum viable model" for those wishing to explore KATs formally or in code.

1.1 What is a KAT?

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

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

To model such a program in KAT, one replaces each concrete test or action with an abstract representation. Let the atomic test α represent the test `i < 50`, β represent `i < 100`, and γ represent `j > 100`; the atomic actions p 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:

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

KATs are naïvely propositional: the algebra understands nothing of the underlying domain or the semantics of the abstract predicates and actions. For example, the fact that $(\mathtt{j} := \mathtt{j} + 2 \cdot \mathtt{j} > 200) \equiv (\mathtt{j} > 198 \cdot \mathtt{j} := \mathtt{j} + 2)$ does not follow from the KAT axioms and must be added manually to any proof as an equational assumption. Yet the ability to reason about the equivalence of programs in the presence of particular domains is critical for reasoning about 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, 30, 40]. Unfortunately, it remains an expert's task to extend the KAT with new domain-specific axioms, provide new proofs of soundness and completeness, and develop the corresponding implementation.

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

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

takes a packet history as input, updates the current packet to have a new destination IP address and port, and then records the current packet state. The original NetKAT paper defines a denotational semantics not just for its primitive parts, but for the various KAT operators; they explicitly restate the KAT equational theory along with custom axioms for the new primitive forms, prove the theory's soundness, and then devise a novel normalization routine to reduce NetKAT to an existing KAT with a known completeness result. Later papers [23, 60] 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 decision procedures when given arbitrary domain-specific theories. Our framework, 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.

We offer some cartoons of KMTs here; see Sec. 2 for technical details.

Consider $\mathrm{P}_{\mathrm{set}}$ (Fig. 1b), a program defined over both naturals and a *set* data structure with two operations: insertion and membership tests. The insertion action $\mathtt{insert}(x, j)$ inserts the value of an expression (j) into a given set (x); the membership test $\mathtt{in}(x, c)$ determines whether a constant

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

(a) $\mathrm{P}_{\mathrm{nat}}$

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

(b) $\mathrm{P}_{\mathrm{set}}$

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

(c) $\mathrm{P}_{\mathrm{map}}$

Fig. 1. Example simple while programs.

(c) is included in a given set (x). An axiom characterizing pushback for this theory has the form:

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

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

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

$$(\mathsf{X}[e_1] := e_2 \cdot \mathsf{X}[c_1] = c_2) \;\equiv\; (e_1 = c_1 \cdot e_2 = c_2 + \mathsf{X}[c_1] = c_2) \cdot \mathsf{X}[e_1] := e_2$$

Each of the theories we have described so far—naturals, sets, booleans, and maps—have tests that only examine the *current* state of the program. However, we need not restrict ourselves in this way. Primitive tests can make dynamic decisions or assertions based on any previous state of the program. As an example, consider the theory of past-time, finite-trace linear temporal logic (LTL_f) [16, 17]. Linear temporal logic introduces new operators such as: $\bigcirc a$ (in the last state a), $\Diamond a$ (in some previous state a), and $\Box a$ (in every state a); we use finite-time LTL because finite traces are a reasonable model in most domains modeling programs.

Finally, we can encode a tracing variant of NetKAT, a system that extends KAT with actions of the form $f \leftarrow v$, where some value v is assigned to one of a finite number of fields f, and tests of the form $f = v$ where field f is tested for value v. It also includes a number of axioms such as $f \leftarrow v \cdot f = v \equiv f \leftarrow v$. The NetKAT axioms can be captured in our framework with minor changes. Further extending NetKAT to Temporal NetKAT is captured trivially in our framework as an application of the LTL_f theory to NetKAT's theory, deriving Beckett et al.'s [8] completeness result compositionally (in fact, we can strengthen it—see Sec. 2.5).

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: we translate programs to a normal form with all predicates at the front (i.e., all predicates become pre-conditions). Pushback generalizes weakest preconditions because we alter the program as we go, possibly changing its commands or structure. Given a primitive action π and a primitive predicate α, the client theory must be able to compute weakest preconditions, telling us how to go from $\pi \cdot \alpha$ to some set of terms: $\sum_{i=0}^{n} \alpha_i \cdot \pi = \alpha_0 \cdot \pi + \alpha_1 \cdot \pi + \dots$. That is, the client theory must be able to take any of its primitive tests and "push it back" through any of its primitive actions. Given the client's notion of weakest preconditions, we can alter programs to take an *arbitrary* term and normalize it into a form where *all* of the predicates appear only at the front of the term, a convenient representation both for our completeness proof (Sec. 3.4) and our implementation (Sec 4).

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. 3). If the pushback has these two properties, we can use it to define a normal form for the KMT generated from the client theory—and we can use that normal form to prove that the resulting KMT is complete and decidable.

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

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

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

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

Syntax

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

Semantics

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

Weakest precondition

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

Axioms

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

Fig. 2. IncNat, increasing naturals

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

For the relative ease of defining IncNat, we get real power—we've extended KAT with unbounded state! It is sound to add other operations to IncNat, like multiplication or addition by a scalar. So long as the operations are monotonically increasing and invertible, we can still define a WP and corresponding axioms. It is *not* possible, however, to compare two variables directly with tests like $x = y$—to do so would not satisfy the requirement that weakest precondition does not grow the size of a test. It would be bad if it did: the test $x = y$ can encode context-free languages! The (inadmissible!) term $x := 0 \cdot y := 0; (\mathsf{inc}_x)^* \cdot (\mathsf{inc}_y)^* \cdot x = y$ describes programs with balanced increments of x and y. For the same reason, we cannot safely add a decrement operation dec_x. Either of these would allow us to define counter machines, leading inevitably to undecidability.

Implementation. Users implement KMT's client theories by defining OCaml modules; users give the types of actions and tests along with functions for parsing, computing subterms, calculating weakest preconditions for primitives, mapping predicates to an SMT solver, and deciding predicate satisfiability (see Sec. 4 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. 2). The user must define two types: a for tests and p for actions. Tests are of the form $x > n$ where variable names are represented with `string`s, and values with OCaml `int`s. Actions hold either the variable being incremented (inc_x) or the variable and value being assigned ($x := n$).

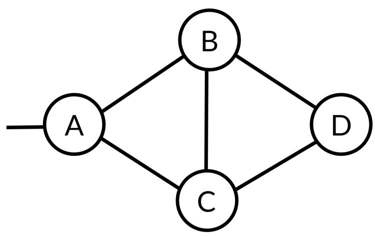
```
type a = Gt of string * int     (* alpha ::= x > n *)
type p = Increment of string   (* pi     ::= inc x *)

module rec IncNat : THEORY with type A.t = a and type P.t = p = struct
  (* generated KMT, for recursive use *)
  module K = KAT (IncNat)
  (* boilerplate necessary for recursive modules, hashconsing *)
  module P : CollectionType with type t = p = struct ... end
  module A : CollectionType with type t = a = struct ... end
  (* extensible parser; pushback; subterms of predicates *)
  let parse name es = ...
  let push_back p a =
    match (p,a) with
    | (Increment _x, Gt (_, j)) when 1 > j → PSet.singleton ~cmp:K.Test.compare (K.one ())
    | (Increment x, Gt (y, j)) when x = y →
      PSet.singleton ~cmp:K.Test.compare (K.theory (Gt (y, j - 1)))
    | (Assign (x,i), Gt (y,j)) when x = y → PSet.singleton ~cmp:K.Test.compare (if i > j then K.one () else
    | _ → PSet.singleton ~cmp:K.Test.compare (K.theory a)
  let rec subterms x =
    match x with
    | Gt (_, 0) → PSet.singleton ~cmp:K.Test.compare (K.theory x)
    | Gt (v, i) → PSet.add (K.theory x) (subterms (Gt (v, i - 1)))
  (* decision procedure for the predicate theory *)
  let satisfiable (a: K.Test.t) = ...
end
```

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

The implementation obligations—syntactic extensions, subterms functions, WP on primitives, a satisfiability checker for the test fragment—mirror our formal development. We offer more client theories in Sec. 2 and more detail on the implementation in Sec. 4.

1.4 A case study: network routing protocols

As a final example demonstrating the kinds of theories supported by KMT, we turn our attention to modeling network routing protocols. While NetKAT uses Kleene algebra to define simple, stateless forwarding tables of networks, the most common network routing protocols are distributed algorithms that actually compute paths in a network by passing messages between devices. As an example the Border Gateway Protocol (BGP) [52], which allows users to define rich routing policy, has become the de facto internet routing protocol used to transport data between between autonomous networks under the control of different entities (e.g. Verizon, Comcast). However, the combination of the distributed nature of BGP, the difficulty of writing policy per-device, and the fact that network devices can and often do fail [28] all contribute to the fact that network outages caused by BGP misconfiguration are common [2, 29, 33, 43, 48, 58, 63]. By encoding BGP policies



prefer neighbor B over A
if from D **then** block **else** accept

(a) Sample network with policy on C

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

(b) P_{SP}, default policy

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

(c) $\mathrm{P}_{\mathrm{BGP}}$, local policies

Fig. 3. An example network and models of BGP routing.

in our framework, it immediately follows that we can decide properties about networks running BGP such as "will router A always be able to reach router B after at most 1 link failure".

Fig. 3a shows an example network that is configured to run BGP. In BGP, devices exchange messages between neighbors to determine routes to a destination. In the figure, router A is connected to an end host (the line going to the left) and wants to tell other routers how to get to this destination.

In the default behavior of the BGP protocol, each router selects the shortest path among all of its neighbors and then informs each of its neighbors about this route (with the path length increased by one). In effect, the devices will compute the shortest paths through the network in a distributed fashion. We can model shortest paths routing in a KMT using the theory of natural numbers: in P_{SP} (Fig. 3b), each router maintains a distance to the destination. Since A knows about the destination, it will start with a distance of 0, while all other routers start with distance ∞. Then, iteratively, each other router updates its distance to be 1 more than the minimum of each of its peers, which is captured by the `min+` operator. The behavior of `min+` can be described by pushback equivalences like:

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

BGP gets interesting when users go beyond shortest path routing and also define router-local policy. In our example network, router C is configured with local policy (Fig. 3a): router C will block messages received from D and will prioritize paths received from neighbor B over those from A (using distance as a tie breaker). In order to accommodate this richer routing behavior, we must extend our model to $\mathrm{P}_{\mathrm{BGP}}$ (Fig. 3c). Now, each router is associated with a variable storing a tuple of both the distance and whether or not the router has a path to the destination; we write C_1 for the "does C have a path" boolean and C_0 for the length of that path, if it exists. We can then create a separate update action for each device in the network to reflect the semantics of the device's local policy (`updateC`, etc.). Further, suppose we have a boolean variable $\mathsf{fail}_{X,Y}$ for each link between routers X and Y indicating whether or not the link is failed. The update action for router C's local policy can be captured with the following type of equivalence:

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

In order for router C to have a path length < 3 to the destination after applying the local update function, it must have either been the case that B did not have a route to the destination (or the B-C link is down) and A had a route with length < 2 and the A-C link is not down, or B had a route with length < 2 and the B-C link is not down. Similarly, we would need an axiom to capture when router C will have a path to the destination based on equivalences like: $\mathtt{updateC} \cdot C_1 \equiv$

$(A_1 \cdot \neg\mathsf{fail}_{A,C} \ + \ B_1 \cdot \neg\mathsf{fail}_{B,C}) \cdot \mathtt{updateC}$—C has a path to the destination if any of its neighbors has a path to the destination and the corresponding link is not failed.

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

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

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

1.5 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. 3). Completeness yields a decision procedure based on normalization.
- Several case studies of this framework (Sec. 2), including a strengthening of Temporal NetKAT's completeness result, theories for unbounded state (naturals, sets, maps), distributed routing protocols, and, most importantly, compositional theories that allow designers to experiment new, complex theories. Several of these theories use unbounded state (e.g., naturals, sets, and maps), going beyond what the state of the art in KAT metatheory is able to accommodate.
- An implementation of KMT (Sec. 4) mirroring our proofs; we derive a normalization-based equivalence decision procedure for client theories from just a few definitions. Our implementation is efficient enough for experimentation with small programs (Sec. 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. 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.

The core technique we discuss here was first developed in Beckett et al.'s work on Temporal NetKAT [8]. Our work here is a significant extension of that work:

- We explicitly define the normalization routine using inference rules (Section 3.3); in Temporal NetKAT, normalization is implicit in its completeness proof.
- The Temporal NetKAT proof of completeness is a morass, simultaneously proving the correctness and termination of normalization. In our framework, we prove those theorems separately (Theorems 3.34 and 3.35).
- Our treatment of negation is improved; we prove a new KAT theorem (Pushback-Neg).
- We present a general *framework* for proving completeness, while the Temporal NetKAT development is specialized to a particular instance—tracing NetKAT with LTL_f.
- The Temporal NetKAT proof achieves only network-wide completeness because of its limited understanding of LTL_f; we are able to achieve completeness.

Beckett et al. handles compilation to forwarding decision diagrams [60], while our presentation doesn't discuss compilation.

Syntax

$$\begin{array}{rcl} \alpha & ::= & b = \mathsf{t} \\ \pi & ::= & b := \mathsf{t} \mid b := \mathsf{f} \\ \mathsf{sub}(\alpha) & = & \{\alpha\} \end{array}$$

Semantics

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

Weakest precondition

$$\begin{array}{l} b := \mathsf{t} \cdot b = \mathsf{t} \text{ WP } 1 \\ b := \mathsf{f} \cdot b = \mathsf{t} \text{ WP } 0 \end{array}$$

Axioms

$$\begin{array}{rl} (b := \mathsf{t}) \cdot (b = \mathsf{t}) \equiv (b := \mathsf{t}) & \textsc{Set-Test-True-True} \\ (b := \mathsf{f}) \cdot (b = \mathsf{t}) \equiv 0 & \textsc{Set-Test-False-True} \end{array}$$

Fig. 4. BitVec, theory of bitvectors

2 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. 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 (Section 3). We start with a simple theory (bit vectors in Sec. 2.1), building up to unbounded state from naturals (Sec. 1.3) to sets and maps parameterized over a notion of value and variable (Sec. 2.3). As an example of a higher-order theory, we define LTL on finite traces (a/k/a LTL_f; Sec. 2.5), extending the predicate language with temporal operators like $\bigcirc a$, meaning "the predicate a holds in the previous state of the trace".

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. 4). The theory adds actions $b := \mathsf{t}$ and $b := \mathsf{f}$ for boolean variables b, and tests of the form $b = \mathsf{t}$, 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: $b = \mathsf{f}$ desugars to $\neg(b = \mathsf{t})$; $\mathsf{flip}\ b$ desugars to $(b = \mathsf{t} \cdot b := \mathsf{f}) + (b = \mathsf{f} \cdot b := \mathsf{t})$. We could go further and define numeric operators on collections of bits, at the cost of producing larger terms. We are not limited to just numbers, of course; once we have bits, we can encode any bounded data structure we like.

KAT+B! [30] develops a nearly identical theory, though our semantics admit different equations. We use a *trace* semantics, where we distinguish between $(b := \mathsf{t} \cdot b := \mathsf{t})$ and $(b := \mathsf{t})$. 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 $(b := \mathsf{t} \cdot b := \mathsf{t}) \equiv (b := \mathsf{t})$. 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. 2.5).

2.2 Disjoint products

Given two client theories, we can combine them into a disjoint product theory, $\mathsf{Prod}(\mathcal{T}_1, \mathcal{T}_2)$, where the 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. 5). We explicitly give definitions for pred and act that defer to the corresponding sub-theory, using t_i to project the trace state to the ith component. It may seem that disjoint products don't give us much, but they in fact allow for us to simulate much more interesting languages in our derived KATs. For example, $\mathsf{Prod}(\mathsf{BitVec}, \mathsf{IncNat})$ allows us to program with both variables valued as either booleans or (increasing) naturals; the product theory lets us directly

Syntax

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

Semantics

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

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

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

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

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

Fig. 5. Prod($\mathcal{T}_1, \mathcal{T}_2$), products of two disjoint theories

Syntax

$$\begin{array}{rcl} & & \alpha ::= x[e] = c \mid e = c \mid \alpha_e \\ & & \pi ::= x[c] := e \mid \pi_e \\ \mathsf{pred}(x[e] = c), t) & = & \mathsf{last}(t)_1(x, \mathsf{last}(t)_2(e)) = c \\ \mathsf{pred}(\alpha_e, t) & = & \mathsf{pred}(\alpha_e, t_2) \\ \mathsf{sub}(x[e] = c) & = & \{x[e] = c\} \cup \\ & & \mathsf{sub}(\neg(e = c')) \\ \mathsf{sub}(e = c) & = & \mathsf{sub}(e = c) \\ \mathsf{sub}(\alpha_e) & = & \mathsf{sub}(\alpha_e) \end{array}$$

Semantics

$$\begin{array}{c} c \in \mathcal{C} \\ e \in \mathcal{E} \\ x \in \mathcal{V} \\ \mathsf{State} = (\mathcal{V} \to \mathcal{C} \to \mathcal{C}) \times (\mathcal{E} \to \mathcal{C}) \\ \mathsf{act}(x[c] := e), \sigma) = \sigma[\sigma_1[x[c \mapsto \sigma_2(e)]]] \\ \mathsf{act}(\pi_e, \sigma) = \sigma[\sigma_2 \mapsto \mathsf{act}(\pi_e, \sigma_2)] \end{array}$$

Pushback extending $\mathcal{E}$

$$\begin{array}{l} (x[c] := e) \cdot \alpha_e \text{ WP } \alpha_e \\ (y[c_1] := e_1) \cdot (x[e_2] = c_2) \text{ WP } x[e_2] = c_2 \\ (x[c_1] := e_1) \cdot (x[e_2] = c_2) \text{ WP } (e_2 = c_1 \cdot e_1 = c_2) + (\neg(e_2 = c_1) \cdot x[e_2] = c_2) \end{array}$$

Axioms extending $\mathcal{E}$

$$\begin{array}{rl} (x[c] := e \cdot \alpha_e) \equiv (\alpha_e \cdot x[c] := e) & \text{E-Comm} \\ (y[c_1] := e_1 \cdot x[e_2] = c_2) \equiv (x[e_2] = c_2 \cdot y[c_1] := e_1) & \text{Map-NEq} \\ (x[c_1] := e_1 \cdot x[e_2] = c_2) \equiv ((e_2 = c_1 \cdot e_1 = c_2) + \neg(e_2 = c_1) \cdot x[e_2] = c_2) \cdot x[c_1] := e_1 & \text{Map-Eq} \end{array}$$

Fig. 6. Map($\mathcal{E}$), unbounded maps over arbitrary expressions/constants

express the sorts of programs that Kozen's early static analysis work had to encode manually, i.e., loops over boolean and numeric state [37].

2.3 Unbounded sets

We define a KMT for unbounded sets parameterized on a theory of expressions $\mathcal{E}$ (Fig. 7). The set data type supports just one operation: $\mathsf{add}(x, e)$ adds the value of expression e to set x (we could add $\mathsf{del}(x, e)$, but we omit it to save space). It also supports a single test: $\mathsf{in}(x, c)$ checks if the constant c is contained in set x. The idea is that $e \in \mathcal{E}$ refers to expressions with, say, variables x and constants c. We allow arbitrary expressions e in some positions and constants c in others. (If we allowed expressions in all positions, WP wouldn't necessarily be non-increasing.)

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

Syntax

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

Semantics

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

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 $\mathcal{E}$

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

Fig. 7. $\mathsf{Set}(\mathcal{E})$, unbounded sets over expressions

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

2.4 Unbounded maps

Maps aren't much different from sets; rather than having simple membership tests, we instead check to see whether a given key maps to a given constant (Fig. 6). Our writes use constant keys and expression values, while our reads use variable keys but constant values. We could have flipped this arrangement—writing to expression keys and reading from constant ones—but we cannot allow *both* reads and writes to expression keys. Doing so would allow us to compare variables, putting us in the realm of context-free languages and foreclosing on the possibility of a complete theory. We could add other operations (at the cost of even more equational rules/pushback entries), like the ability to remove keys from maps or to test whether a key is in the map or not. Just as for $\mathsf{Set}(\mathcal{E})$, we must put all $x[e] = c$ and $x[e] \neq c$ formulae *later* in the global well ordering than any of those generated by the client theory's $e = c$ or $e \neq c$.

2.5 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. 8). For information on LTL_f, we refer the reader to work by Baier and McIlraith, De Giacomo and Vardi, Roşu, and Beckett et al., and Campbell and Greenberg [5, 8, 10, 11, 16, 17, 53].

LTL_f adds just two predicates: $\bigcirc a$, pronounced "last a", means a held in the prior state; and $a \mathcal{S} b$, pronounced "a since b", means b held at some point in the past, and a has held since then. There is a slight subtlety around the beginning of time: we say that $\bigcirc a$ is false at the beginning (what can be true in a state that never happened?), and $a \mathcal{S} b$ degenerates to b at the beginning of time. The last and since predicates together are enough to encode the rest of LTL_f; encodings are given below the syntax. The pred definitions mostly defer to the client theory's definition of pred (which may

Syntax

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

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

Semantics

$$\begin{aligned}
\mathsf{State} &= \mathsf{State}_{\mathcal{T}} \\
\mathsf{pred}(\bigcirc a, \langle \sigma, l \rangle) &= \mathfrak{f} \\
\mathsf{pred}(\bigcirc a, t\langle \sigma, l \rangle) &= \mathsf{pred}(a, t) \\
\mathsf{pred}(a \mathrel{\mathcal{S}} b, \langle \sigma, l \rangle) &= \mathsf{pred}(b, \langle \sigma, l \rangle) \\
\mathsf{pred}(a \mathrel{\mathcal{S}} b, t\langle \sigma, l \rangle) &= \mathsf{pred}(b, t\langle \sigma, l \rangle) \vee \\
&\quad (\mathsf{pred}(a, t\langle \sigma, l \rangle) \wedge \mathsf{pred}(a \mathrel{\mathcal{S}} b, t))
\end{aligned}$$

Weakest precondition extending $\mathcal{T}$

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

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

Axioms extending $\mathcal{T}$

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

Fig. 8. $\mathrm{LTL}_f(\mathcal{T})$, linear temporal logic on finite traces over an arbitrary theory

recursively reference the LTL_f pred function), unrolling $\mathcal{S}$ as it goes (LTL-Since-Unroll). Weakest preconditions uses inference rules: to push back $\mathcal{S}$, we unroll $a \mathrel{\mathcal{S}} b$ into $a \cdot \bigcirc(a \mathrel{\mathcal{S}} b) + b$; pushing last through an action is easy, but pushing back a or b recursively uses the $\mathsf{PB}^{\bullet}$ judgment. Adding these rules leaves our judgments monotonic, and if $\pi \cdot a \ \mathsf{PB}^{\bullet}\ x$, then $x = \sum a_i \pi$(Lemma 3.33). 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 Roşu's proof with coinductive axioms [53].

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

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

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

The ability to use temporal logic in KAT means that we can model check programs by phrasing model checking questions in terms of program equivalence. For example, for some program r, we

Syntax

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

Semantics

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

Weakest precondition

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

Axioms

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

Fig. 9. Tracing NetKAT a/k/a NetKAT without dup

can check if $r \equiv r \cdot \bigcirc\Box(j \leq 200)$. In other words, if there exists some program trace that does not satisfy the test, then it will be filtered—resulting in non-equivalent terms. If the terms are equal, then every trace from r satisfies the test. Similarly, we can test whether $r \cdot \bigcirc\Box(j \leq 200)$ is empty—if so, there are *no* satisfying traces.

In addition to model checking, temporal logic is a useful programming language feature: programs can make dynamic program decisions based on the past more concisely. Such a feature is useful for Temporal NetKAT (Sec. 2.7 below), but could also be used for, e.g., regular expressions with lookbehind or even a limited form of back-reference.

2.6 Tracing NetKAT

We define NetKAT as a KMT over packets, which we model as functions from packet fields to values (Fig. 9). KMT's trace semantics diverge slightly from NetKAT's: like KAT+B! (Sec. 2.1; [30]), 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. Typically, dups are automatically inserted at the topology level, i.e., before a packet enters a switch, we record its state by running dup. From our perspective, NetKAT very nearly has a tracing semantics, but the traces are selective. If we put an implicit dup before *every* field update, NetKAT has our tracing semantics. The upshot is that our "tracing NetKAT" has a slightly different equational theory from conventional NetKAT, rejecting the following NetKAT laws as unsound for trace semantics:

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

In principle, one can abstract our semantics' traces to find the more restricted NetKAT traces, but we can't offer any formal support in our framework for abstracted reasoning. Just as for BitVec, It is possible that ideas from Kozen and Mamouras could apply here [40]; see Sec. 6.

2.7 Temporal NetKAT

We derive Temporal NetKAT as LTL_f(NetKAT), i.e., LTL_f instantiated over tracing NetKAT; the combination yields precisely the system described in the Temporal NetKAT paper [8]. Our LTL_f theory can now rely on Campbell and Greenberg's proof of deductive completeness for LTL_f [10, 11], we can automatically derive a stronger completeness result for Temporal NetKAT than that from

Syntax

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

Semantics

$$
\begin{array}{rcl}
n & \in & \mathbb{N} \cup \{\infty\} \\
x & \in & \mathcal{V} \\
\mathsf{State} & = & \mathcal{V} \to \mathbb{N}
\end{array}
$$

Weakest precondition axioms are identical to pushback

$$
\begin{array}{l}
x := \mathsf{min+}(\vec{x}) \cdot (x < \infty) \ \mathsf{WP}\ \Sigma_i (x_i < \infty) \\
x := \mathsf{min+}(\vec{x}) \cdot (x < n) \ \mathsf{WP}\ \Sigma_i (x_i < n - 1)
\end{array}
$$

Fig. 10. SP, shortest paths in a graph

Syntax

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

Semantics

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

Weakest precondition axioms are identical to pushback

$$
\begin{array}{lll}
(\pi \cdot \mathsf{fail}_{R_1,R_2}) & \mathsf{WP} & \mathsf{fail}_{R_1,R_2} \\
(\mathsf{updateC} \cdot D_1) & \mathsf{WP} & D_1 \\
(\mathsf{updateC} \cdot C_1) & \mathsf{WP} & \neg\mathsf{fail}_{A,C} \cdot A_1 + \neg\mathsf{fail}_{B,C} \cdot B_1 \\
(\mathsf{updateC} \cdot D_0 < n) & \mathsf{WP} & (D_0 < n) \\
(\mathsf{updateC} \cdot C_0 < n) & \mathsf{WP} & \neg\mathsf{fail}_{A,C} \cdot (\neg B_1 + \mathsf{fail}_{B,C}) \cdot A_1 \cdot (A_0 < n-1) + \\
& & \neg\mathsf{fail}_{B,C} \cdot B_1 \cdot (B_0 < n-1)
\end{array}
$$

Fig. 11. BGP, protocol theory for router C from the network in Fig. 3

the paper, which showed completeness only for "network-wide" policies, i.e., those with start at the front.

2.8 Distributed routing protocols

The theory for naturals with the min+ operator used for shortest path routing is shown in Fig. 10. The theory is similar to the `IncNat` theory but for some minor differences. First, the domain is now over $\mathbb{N} \cup \{\infty\}$. Second, there is a new axiom and pushback relation relating min+ to a test of the form $x < n$. Third, the subterms function is now defined in terms of all other variables, which are infinite in principle but finite in any given term (e.g., the number of routers in a given network).

The theory for the BGP protocol instance with local router policy described in Fig. 3 is now shown in Fig. 11. For brevity, we only show the theory for router C in the network. The state has two parts: the first part maps each router to a pair of a natural number describing the path length to the destination for that router, and a boolean describing whether or not the router has a route

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

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

Actions

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

Fig. 12. $\mathcal{T}^*$: generalized KAT syntax over a client theory $\mathcal{T}$ (client parts highlighted)

to the destination; the second part maps links to a boolean representing whether the link is up or not. We require new axioms corresponding to each of the pushback operations shown. The action updateC commutes with unrelated tests, and otherwise behaves as described in Sec. 1.

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 mathematics to use our framework—we do it once and for all, so you don't have to. We first explain the structure of our framework for defining a KAT in terms of a client theory. While we have striven to make this section accessible to non-expert readers, those completely new to KATs may do well to skip our discussion of pushback (Sec. 3.3.2 on) and read our case studies (Sec. 2). We highlight anything the client theory must provide.

We derive a KAT $\mathcal{T}^*$ (Fig. 12) on top of a client theory $\mathcal{T}$ where $\mathcal{T}$ has two fundamental parts—predicates $\alpha \in \mathcal{T}_\alpha$ and actions $\pi \in \mathcal{T}_\pi$. These are the *primitives* of the client theory. We refer to the Boolean algebra over the client theory as $\mathcal{T}^*_{\text{pred}} \subseteq \mathcal{T}^*$.

Our framework can provide results for $\mathcal{T}^*$ in a pay-as-you-go fashion: given a notion of state and an interpretation for the predicates and actions of $\mathcal{T}$, we derive a trace semantics for $\mathcal{T}^*$ (Sec. 3.1); if $\mathcal{T}$ has a sound equational theory with respect to our semantics, so does $\mathcal{T}^*$ (Sec. 3.2); if $\mathcal{T}$ has a complete equational theory with respect to our semantics, and satisfies certain weakest precondition requirements, then $\mathcal{T}^*$ has a complete equational theory (Sec. 3.4); and finally, with just a few lines of code defining the structure of $\mathcal{T}$, we can provide a decision procedure for equivalence (Sec. 4)using the normalization routine from completeness (Sec. 3.4).

The key to our general, parameterized proof is a novel *pushback* operation that generalizes weakest preconditions (Sec. 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 (Sec. 3.4).

3.1 Semantics

The first step in turning the client theory $\mathcal{T}$ into a KAT is to define a semantics (Fig. 13). We can give any KAT a *trace semantics*: the meaning of a term is a trace t, which is a non-empty list of log entries l. Each *log entry* records a state σ and (in all but the initial state) a primitive action π. The client assigns meaning to predicates and actions by defining a set of states State and two functions: one to determine whether a predicate holds (pred) and another to determine an action's effects (act). To run a $\mathcal{T}^*$ term on a state σ, we start with an initial state $\langle \sigma, \bot \rangle$; when we're done, we'll have a set of traces of the form $\langle \sigma_0, \bot \rangle \langle \sigma_1, \pi_1 \rangle \ldots$, where $\sigma_i = \mathsf{act}(\pi_i, \sigma_{i-1})$ for $i > 0$. (A similar semantics shows up in Kozen's application of KAT to static analysis [37].)

A reader new to KATs should compare this definition with that of NetKAT or Temporal NetKAT [1, 8]: defined recursively over the syntax, the denotation function collapses predicates and actions

Trace definitions

$$
\begin{array}{lcll}
\sigma & \in & \mathsf{State} & \\
l & \in & \mathsf{Log} & ::= \langle \sigma, \bot \rangle \mid \langle \sigma, \pi \rangle \\
t & \in & \mathsf{Trace} & = \mathsf{Log}^{+}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{pred} & : & \mathcal{T}_\alpha \times \mathsf{Trace} \to \{\mathfrak{t}, \mathfrak{f}\} \\
\mathsf{act} & : & \mathcal{T}_\pi \times \mathsf{State} \to \mathsf{State}
\end{array}
$$

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

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

$$(f \bullet g)(t) = \bigcup_{t' \in f(t)} g(t')$$

$$f^0(t) = \{t\} \qquad f^{i+1}(t) = (f \bullet f^i)(t)$$

$$\mathsf{last}(\ldots \langle \sigma, \_ \rangle) = \sigma$$

Kleene Algebra axioms

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

$$p \leq q \Leftrightarrow p + q \equiv q$$

Boolean Algebra axioms

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

Consequences

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

Fig. 13. Semantics and equational theory for $\mathcal{T}^*$

into a single semantics, using Kleisli composition (written $\bullet$) to give meaning to sequence and an infinite union and exponentiation (written $-^n$) to give meaning to Kleene star. We've generalized the way that predicates and actions work, though, deferring to two functions that must be defined by the client theory: pred and act.

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. (Composite predicates follow this same pattern, always returning either a singleton set holding their input trace or the empty set(Lemma 3.4).) It's acceptable for the pred function to recursively call the denotational semantics, though we have skipped the formal detail here. This way we can define composite primitive predicates as in, e.g., temporal logic (Sec. 2.7).

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

3.2 Soundness

Proving 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. 13) even mention the client's primitives. Pushback negation is a novel KAT theorem (Pushback-Neg); it generalizes the result that theorem that $b \cdot p \equiv p \cdot b \leftrightarrow b \cdot p \cdot \neg b + \neg b \cdot p \cdot b \equiv 0$ from Kozen [36].

Lemma 3.1 (Pushback negation (Pushback-Neg)). *$p \cdot a \equiv b \cdot p$ iff $p \cdot \neg a \equiv \neg b \cdot p$.*

Proof. We show that both sides $p \cdot \neg a$ and $\neg b \cdot p$ are equivalent to $\neg b \cdot p \cdot \neg a$ by way of BA-Excl-Mid:

$$\begin{array}{rll}
p \cdot \neg a \equiv & (b + \neg b) \cdot p \cdot \neg a & \text{(KA-Seq-One, BA-Excl-Mid)} \\
\equiv & b \cdot p \cdot \neg a + \neg b \cdot p \cdot \neg a & \text{(KA-Dist-L)} \\
\equiv & p \cdot a \cdot \neg a + \neg b \cdot p \cdot \neg a & \text{(assumption)} \\
\equiv & p \cdot 0 + \neg b \cdot p \cdot \neg a & \text{(BA-Contra)} \\
\equiv & \neg b \cdot p \cdot \neg a & \text{(KA-Plus-Comm, KA-Plus-Zero)} \\
\equiv & 0 \cdot p + \neg b \cdot p \cdot \neg a & \text{(BA-Contra)} \\
\equiv & \neg b \cdot b \cdot p + \neg b \cdot p \cdot \neg a & \text{(assumption)} \\
\equiv & \neg b \cdot p \cdot a + \neg b \cdot p \cdot \neg a & \text{(KA-Dist-R)} \\
\equiv & \neg b \cdot p \cdot (a + \neg a) & \text{(KA-One-Seq, BA-Excl-Mid)} \\
\equiv & \neg b \cdot p & \square
\end{array}$$

The other direction of the proof is symmetric, with the two terms meeting at $b \cdot p \cdot a$.

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. 13, 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.

Lemma 3.2 (Kleisli composition is associative). $[\![p]\!] \bullet ([\![q]\!] \bullet [\![r]\!]) = ([\![p]\!] \bullet [\![q]\!]) \bullet [\![r]\!]$.

Proof. By direct computation. $\square$

Lemma 3.3 (Exponentiation commutes). $[\![p]\!]^{i+1} = [\![p]\!]^{i} \bullet [\![p]\!]$

Proof. By induction on i. When $i = 0$, both yield $[\![p]\!]$. In the inductive case, we compute:

$$\begin{array}{rll}
[\![p]\!]^{i+2} = & [\![p]\!] \bullet [\![p]\!]^{i+1} & \\
= & [\![p]\!] \bullet ([\![p]\!]^{i} \bullet [\![p]\!]) & \text{by the IH} \\
= & ([\![p]\!] \bullet [\![p]\!]^{i}) \bullet [\![p]\!] & \text{by Lemma 3.2} \\
= & ([\![p]\!]^{i+1}) \bullet [\![p]\!] & \text{by Lemma 3.2}
\end{array}$$

$\square$

Lemma 3.4 (Predicates produce singleton or empty sets). $[\![a]\!](t) \subseteq \{t\}$.

Proof. By induction on a, leaving t general. $\square$

Theorem 3.5 (Soundness of $\mathcal{T}^*$). *If $\mathcal{T}$'s equational reasoning is sound ($p \equiv_{\mathcal{T}} q \Rightarrow [\![p]\!] = [\![q]\!]$) then $\mathcal{T}^*$'s equational reasoning is sound ($p \equiv q \Rightarrow [\![p]\!] = [\![q]\!]$).*

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

(KA-Plus-Assoc) We have $p + (q + r) \equiv (p + q) + r$; by associativity of union.

(KA-Plus-Comm) We have $p + q \equiv q + p$; by commutativity of union.

(KA-Plus-Zero) We have $p + 0 \equiv p$; immediate, since $[\![0]\!](t) = \emptyset$.

(KA-Plus-Idem) By idempotence of union $p + p \equiv p$.

(KA-Seq-Assoc) We have $p \cdot (q \cdot r) \equiv (p \cdot q) \cdot r$; by Lemma 3.2.

(KA-Seq-One) We have $1 \cdot p \equiv p$; immediate, since $[\![1]\!](t) = \{t\}$.

(KA-One-Seq) We have $p \cdot 1 \equiv p$; immediate, since $[\![1]\!](t) = \{t\}$.

(KA-Dist-L) We have $p \cdot (q + r) \equiv p \cdot q + p \cdot r$; we compute:

$$\begin{aligned}
[\![p \cdot (q + r)]\!](t) &= \textstyle\bigcup_{t' \in [\![p]\!](t)} [\![q + r]\!](t') \\
&= \textstyle\bigcup_{t' \in [\![p]\!](t)} [\![q]\!](t') \cup [\![r]\!](t') \\
&= \textstyle\bigcup_{t' \in [\![p]\!](t)} [\![q]\!](t') \cup \bigcup_{t' \in [\![p]\!](t)} [\![r]\!](t') \\
&= [\![p \cdot q]\!](t) \cup [\![p \cdot r]\!](t) \\
&= [\![p \cdot q + p \cdot r]\!](t)
\end{aligned}$$

(KA-Dist-R) As for KA-Dist-L.

(KA-Zero-Seq) We have $0 \cdot p \equiv 0$; immediate, since $[\![0]\!](t) = \emptyset$.

(KA-Seq-Zero) We have $p \cdot 0 \equiv 0$; immediate, since $[\![0]\!](t) = \emptyset$.

(KA-Unroll-L) We have $p^* \equiv 1 + p \cdot p^*$. We compute:

$$\begin{aligned}
[\![p^*]\!](t) &= \textstyle\bigcup_{0 \le i} [\![p]\!]^i(t) \\
&= [\![1]\!](t) \cup \textstyle\bigcup_{1 \le i} [\![p]\!]^i(t) \\
&= [\![1]\!](t) \cup [\![p]\!](t) \cup \textstyle\bigcup_{2 \le i} [\![p]\!]^i(t)) \\
&= [\![1]\!](t) \cup ([\![p]\!] \bullet [\![1]\!])(t) \cup \textstyle\bigcup_{1 \le i} ([\![p]\!] \bullet [\![p]\!]^i)(t) \\
&= [\![1]\!](t) \cup ([\![p]\!] \bullet [\![1]\!])(t') \cup ([\![p]\!] \bullet \textstyle\bigcup_{1 \le i} [\![p]\!]^i)(t) \\
&= [\![1]\!](t) \cup ([\![p]\!] \bullet \textstyle\bigcup_{0 \le i} [\![p]\!]^i)(t) \\
&= [\![1]\!](t) \cup [\![p \cdot p^*]\!](t)) \\
&= [\![1 + p \cdot p^*]\!](t)
\end{aligned}$$

(KA-Unroll-R) As for KA-Unroll-L.

(KA-LFP-L) We have $p^* \cdot q \le r$, i.e., $p^* \cdot q + r \equiv r$. By the IH, we know that $[\![q]\!](t) \cup ([\![p]\!] \bullet [\![r]\!])(t) \cup [\![r]\!](t) = [\![r]\!](t)$. We show, by induction on i, that $([\![p]\!]^i \bullet [\![q]\!])(t) \cup [\![r]\!](t) = [\![r]\!](t)$.

$(i = 0)$ We compute:

$$\begin{aligned}
&([\![p]\!]^0 \bullet [\![q]\!])(t) \cup [\![r]\!](t) \\
=\ &([\![1]\!] \bullet [\![q]\!])(t) \cup [\![r]\!](t) \\
=\ &[\![q]\!](t) \cup [\![r]\!](t) \\
=\ &[\![q]\!](t) \cup ([\![q]\!](t) \cup ([\![p]\!] \bullet [\![r]\!])(t) \cup [\![r]\!]) \quad \text{by the outer IH} \\
=\ &[\![q]\!](t) \cup ([\![p]\!] \cdot [\![r]\!])(t) \cup [\![r]\!](t) \\
=\ &[\![r]\!](t) \quad \text{by the outer IH again}
\end{aligned}$$

$(i = i' + 1)$ We compute:

$$\begin{aligned}
&([\![p]\!]^{i'+1} \bullet [\![q]\!])(t) \cup [\![r]\!](t) \\
=\ &([\![p]\!] \bullet [\![p]\!]^{i'} \bullet [\![q]\!])(t) \cup [\![r]\!](t) \\
=\ &([\![p]\!] \bullet [\![p]\!]^{i'} \bullet [\![q]\!])(t) \cup ([\![q]\!](t) \cup ([\![p]\!] \bullet [\![r]\!])(t) \cup [\![r]\!](t)) \quad \text{by the outer IH} \\
=\ &\textstyle\bigcup_{t' \in [\![p]\!](t)} (\bigcup_{t'' \in [\![p]\!]^{i'}(t')} [\![q]\!](t') \cup [\![r]\!](t')) \cup ([\![q]\!](t) \cup [\![r]\!](t)) \\
=\ &([\![p]\!] \bullet [\![r]\!])(t) \cup ([\![q]\!](t) \cup [\![r]\!](t)) \quad \text{by the inner IH} \\
=\ &[\![r]\!](t) \quad \text{by the outer IH again}
\end{aligned}$$

So, finally, we have:

$$[\![p^* \cdot q + r]\!](t) = (\bigcup_{0 \le i} [\![p]\!]^i \bullet [\![q]\!])(t) \cup [\![r]\!](t) = \bigcup_{0 \le i} ([\![p]\!]^i \bullet [\![q]\!])(t) \cup [\![r]\!](t)) = \bigcup_{0 \le i} [\![r]\!](t) = [\![r]\!](t)$$

(KA-LFP-R) As for KA-LFP-L.

(BA-Plus-Dist) We have $a + (b \cdot c) \equiv (a + b) \cdot (a + c)$. We have $[\![a + (b \cdot c)]\!](t) = [\![a]\!](t) \cup ([\![b]\!] \bullet [\![c]\!])(t)$. By Lemma 3.4, we know that each of these denotations produces either $\{t\}$ or $\emptyset$, where $\cup$ is disjunction and $\bullet$ is conjunction. By distributivity of these operations.

(BA-Plus-One) We have $a + 1 \equiv 1$; we have this directly by Lemma 3.4.

(BA-Excl-Mid) We have $a + \neg a \equiv 1$; we have this directly by Lemma 3.4 and the definition of negation.

(BA-Seq-Comm) $a \cdot b \equiv b \cdot a$; we have this directly by Lemma 3.4 and unfolding the union.

(BA-Contra) We have $a \cdot \neg a \equiv 0$; we have this directly by Lemma 3.4 and the definition of negation.

(BA-Seq-Idem) $a \cdot a \equiv a$; we have this directly by Lemma 3.4 and unfolding the union.

□

For the duration of Sec. 3, we assume that any equations $\mathcal{T}$ adds are sound and, so, $\mathcal{T}^*$ is sound by Theorem 3.5.

3.3 Normalization via pushback

In order to prove completeness (Sec. 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. 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. 3.3.1); and
(2) weakest preconditions for primitives (Sec. 3.3.2).

Once we've defined our normalization procedure, we can use it prove completeness (Sec. 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 a complex one. For example, pushing a predicate back may not eliminate the predicate even though progress was made in getting predicates to the front. More trickily, it may be that pushing test a back through π yields $\sum a_i \cdot \pi$ where each of the a_i is a copy of some subterm of a—and there may be *many* such copies!

Let the set of *restricted actions* $\mathcal{T}_{\mathsf{RA}}$ be the subset of $\mathcal{T}^*$ where the only test is 1. We will use metavariables m, n, and l to denote elements of $\mathcal{T}_{\mathsf{RA}}$. Let the set of *normal forms* $\mathcal{T}^*_{\mathsf{nf}}$ be a set of pairs of tests $a_i \in \mathcal{T}^*_{\mathsf{pred}}$ and restricted actions $m_i \in \mathcal{T}_{\mathsf{RA}}$. We will use metavariables t, u, v, w, x, y, and z to denote elements of $\mathcal{T}^*_{\mathsf{nf}}$; we typically write these sets not in set notation, but as sums, i.e., $x = \sum_{i=1}^{k} a_i \cdot m_i$ means $x = \{(a_1, m_1), (a_2, m_2), \ldots, (a_k, m_k)\}$. The sum notation is convenient, but it is important that normal forms really be treated as sets—there should be no duplicated terms in the sum. We write $\sum_i a_i$ to denote the normal form $\sum_i a_i \cdot 1$. We will call a normal form *vacuous* when it is the empty set (i.e., the empty sum, which we interpret conventionally as 0) or when

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

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

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

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

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

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

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

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

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

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

Fig. 14. Maximal tests and the maximal subterm ordering

all of its tests are 0. The set of normal forms, $\mathcal{T}^*_{\mathrm{nf}}$, 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 $\mathcal{T}^*_{\mathrm{nf}}$.

The definitions for the maximal subterm ordering are complex (Fig. 14), but the intuition is: seqs gets all the tests out of a predicate; tests gets all the predicates out of a normal form; sub gets subterms; mt gets "maximal" tests that cover a whole set of tests; we lift mt to work on normal forms by extracting all possible tests; the relation $x \preceq y$ means that y's maximal tests include all of x's maximal tests. Maximal tests indicate which test to push back next in order to make progress towards normalization. For example, the subterms of $\Diamond x > 1$ are defined by the client theory (Sec. 2.5) as $\{\Diamond x > 1, x > 1, x > 0, 1, 0\}$, which represents the possible tests that might be generated pushing back $\Diamond x > 1$; the maximal tests of $\Diamond x > 1$ are just $\{\Diamond x > 1\}$; the maximal tests of the set $\{\Diamond x > 1, x > 0, y > 6\}$ are $\{\Diamond x > 1, y > 6\}$ since these tests are not subterms of any other test. Therefore, we can choose to push back either of $\Diamond x > 1$ or $y > 6$ next and know that we will continue making progress towards normalization.

Lemma 3.6 (Terms are subterms of themselves). $a \in \mathsf{sub}(a)$

Proof. By induction on a. All cases are immediate except for $\neg a$, which uses the IH. □

Lemma 3.7 (0 is a subterm of all terms). $0 \in \mathsf{sub}(a)$

Proof. By induction on a. The cases for 0, 1, and α are immediate; the rest of the cases follow by the IH. □

Lemma 3.8 (Maximal tests are tests). $\mathsf{mt}(A) \subseteq \mathsf{seqs}(A)$ *for all sets of tests A.*

Proof. We have by definition:

$$\begin{array}{rcl} \mathsf{mt}(A) & = & \{b \in \mathsf{seqs}(A) \mid \forall c \in \mathsf{seqs}(A),\ c \neq b \Rightarrow b \notin \mathsf{sub}(c)\} \\ & \subseteq & \mathsf{seqs}(A) \end{array}$$

□

Lemma 3.9 (Maximal tests contain all tests). $\mathsf{seqs}(A) \subseteq \mathsf{sub}(\mathsf{mt}(A))$ *for all sets of tests* A.

Proof. Let an $a \in \mathsf{seqs}(A)$ be given; we must show that $a \in \mathsf{sub}(\mathsf{mt}(A))$. If $a \in \mathsf{mt}(A)$, then $a \in \mathsf{sub}(\mathsf{mt}(A))$ (Lemma 3.6). If $a \notin \mathsf{mt}(A)$, then there must exist a $b \in \mathsf{mt}(A)$ such that $a \in \mathsf{sub}(b)$. But in that case, $a \in \mathsf{sub}(b) \cup \bigcup_{a \in \mathsf{mt}(A) \setminus \{b\}} \mathsf{sub}(\mathsf{mt}(a))$, so $a \in \mathsf{sub}(\mathsf{mt}(A))$. □

Lemma 3.10 (seqs distributes over union). $\mathsf{seqs}(A \cup B) = \mathsf{seqs}(A) \cup \mathsf{seqs}(B)$

Proof. We compute:

$$\begin{array}{rcl} \mathsf{seqs}(A \cup B) & = & \bigcup_{c \in A \cup B} \mathsf{seqs}(c) \\ & = & \bigcup_{c \in A} \mathsf{seqs}(c) \cup \bigcup_{c \in B} \mathsf{seqs}(c) \\ & = & \mathsf{seqs}(A) \cup \mathsf{seqs}(B) \end{array}$$

□

Lemma 3.11 (seqs is idempotent). $\mathsf{seqs}(a) = \mathsf{seqs}(\mathsf{seqs}(a))$

Proof. By induction on a. □

We can lift Lemma 3.11 to sets of terms, as well.

Lemma 3.12 (Sequence extraction). *If* $\mathsf{seqs}(a) = \{a_1, \ldots, a_k\}$ *then* $a \equiv a_1 \cdot \ldots \cdot a_k$.

Proof. By induction on a. The only interesting case is when $a = b \cdot c$. We have:

$$\{a_1, \ldots, a_k\} = \mathsf{seqs}(a) = \mathsf{seqs}(b \cdot c) = \mathsf{seqs}(b) \cup \mathsf{seqs}(c).$$

Furthermore, $\mathsf{seqs}(b)$ (resp. $\mathsf{seqs}(c)$) is equal to some subset of the $a_i \in \mathsf{seqs}(a)$, such that $\mathsf{seqs}(b) \cup \mathsf{seqs}(c) = \mathsf{seqs}(a)$. By the IH, we know that $b \equiv \Pi_{b_i \in \mathsf{seqs}(b)} b_i$ and $c \equiv \Pi_{c_i \in \mathsf{seqs}(c)} c_i$, so we have:

$$\begin{array}{rll} a & \equiv b \cdot c & \\ & \equiv \left(\prod_{b_i \in \mathsf{seqs}(b)} b_i\right) \cdot \left(\prod_{b_i \in \mathsf{seqs}(b)} b_i\right) & (\textsc{BA-Seq-Idem}) \\ & \equiv \prod_{a_i \in \mathsf{seqs}(b) \cup \mathsf{seqs}(c)} a_i & (\textsc{BA-Seq-Comm}) \\ & \equiv \prod_{i=1}^{k} a_i & \end{array}$$

□

Corollary 3.13 (Maximal tests are invariant over tests). $\mathsf{mt}(A) = \mathsf{mt}(\mathsf{seqs}(A))$

Proof. We compute:

$$\begin{array}{rcl} \mathsf{mt}(A) & = & \{b \in \mathsf{seqs}(A) \mid \forall c \in \mathsf{seqs}(A), c \neq b \Rightarrow b \notin \mathsf{sub}(c)\} \\ & & \hfill (\text{Lemma 3.11}) \\ & = & \{b \in \mathsf{seqs}(\mathsf{seqs}(A)) \mid \forall c \in \mathsf{seqs}(\mathsf{seqs}(A)), c \neq b \Rightarrow b \notin \mathsf{sub}(c)\} \\ & = & \mathsf{mt}(\mathsf{seqs}(A)) \end{array}$$

□

Lemma 3.14 (Subterms are closed under subterms). *If* $a \in \mathsf{sub}(b)$ *then* $\mathsf{sub}(a) \subseteq \mathsf{sub}(b)$.

Proof. By induction on b, letting some $a \in \mathsf{sub}(b)$ be given. □

Lemma 3.15 (Subterms decrease in size). *If $a \in \mathsf{sub}(b)$, then either $a \in \{0, 1, b\}$ or a comes before b in the global well ordering.*

Proof. By induction on b. □

Lemma 3.16 (Maximal tests always exist). *If A is a non-empty set of tests, then* $\mathsf{mt}(A) \neq \emptyset$.

Proof. We must show there exists at least one term in $\mathsf{mt}(A)$.

If $\mathsf{seqs}(A) = \{a\}$, then a is a maximal test. If $\mathsf{seqs}(A) = \{0, 1\}$, then 1 is a maximal test. If $\mathsf{seqs}(A) = \{0, 1, \alpha\}$, then α is a maximal test. If $\mathsf{seqs}(A)$ isn't any of those, then let $a\mathsf{seqs}A$ be the term that comes last in the well ordering on predicates.

To see why $a \in mt(A)$, suppose (for a contradiction) we have $b \in \mathsf{mt}(A)$ such $b \neq a$ and $a \in \mathsf{sub}(b)$. By Lemma 3.15, either $a \in \{0, 1, b\}$ or a comes before b in the global well ordering. We've ruled out the first two cases above. If $a = b$, then we're fine—a is a maximal test. But if a comes before b in the well ordering, we've reached a contradiction, since we selected a as the term which comes *latest* in the well ordering. □

As a corollary, note that a maximal test exists even for vacuous normal forms, where $\mathsf{mt}(x) = \{0\}$ when x is vacuous.

Lemma 3.17 (Maximal tests generate subterms). $\mathsf{sub}(\mathsf{mt}(A)) = \bigcup_{a \in \mathsf{seqs}(A)} \mathsf{sub}(a)$

Proof. Since $\mathsf{mt}(A) \subseteq \mathsf{seqs}(A)$ (Lemma 3.8), we can restate our goal as:

$$\mathsf{sub}(\mathsf{mt}(A)) = \bigcup_{a \in \mathsf{mt}(A)} \mathsf{sub}(a) \cup \bigcup_{a \in \mathsf{seqs}(A) \setminus \mathsf{mt}(A)} \mathsf{sub}(a)$$

We have $\mathsf{sub}(\mathsf{mt}(A)) = \bigcup_{a \in \mathsf{mt}(A)} \mathsf{sub}(a)$ by definition; it remains to see that the latter union is subsumed by the former; but we have $\mathsf{seqs}(A) \subseteq \mathsf{sub}(\mathsf{mt}(A))$ by Lemma 3.9. □

Lemma 3.18 (Union distributes over maximal tests).
$\mathsf{sub}(\mathsf{mt}(A \cup B)) = \mathsf{sub}(\mathsf{mt}(A)) \cup \mathsf{sub}(\mathsf{mt}(B))$

Proof. We compute:

$$\begin{aligned}
\mathsf{sub}(\mathsf{mt}(A \cup B)) &= \textstyle\bigcup_{a \in \mathsf{seqs}(A \cup B)} \mathsf{sub}(a) && \text{(Lemma 3.17)} \\
&= \textstyle\bigcup_{a \in \mathsf{seqs}(A) \cup \mathsf{seqs}(B)} \mathsf{sub}(a) \\
&= \left[\textstyle\bigcup_{a \in \mathsf{seqs}(A)} a\right] \cup \left[\textstyle\bigcup_{b \in \mathsf{seqs}(B)} \mathsf{sub}(b)\right] \\
&= \mathsf{sub}(\mathsf{mt}(A)) \cup \mathsf{sub}(\mathsf{mt}(B)) && \text{(Lemma 3.17)}
\end{aligned}$$

□

Lemma 3.19 (Maximal tests are monotonic). *If $A \subseteq B$ then* $\mathsf{sub}(\mathsf{mt}(A)) \subseteq \mathsf{sub}(\mathsf{mt}(B))$.

Proof. We have $\mathsf{sub}(\mathsf{mt}(B)) = \mathsf{sub}(\mathsf{mt}(A \cup B)) = \mathsf{sub}(\mathsf{mt}(A)) \cup \mathsf{sub}(\mathsf{mt}(B))$ (by Lemma 3.18). □

Corollary 3.20 (Sequences of maximal tests). $\mathsf{sub}(\mathsf{mt}(a \cdot b)) = \mathsf{sub}(\mathsf{mt}(a)) \cup \mathsf{sub}(\mathsf{mt}(b))$

Proof.

$$\begin{aligned}
&\mathsf{sub}(\mathsf{mt}(c \cdot d)) \\
= \;&\mathsf{sub}(\mathsf{mt}(\mathsf{seqs}(c \cdot d))) && \text{(Corollary 3.13)} \\
= \;&\mathsf{sub}(\mathsf{mt}(\mathsf{seqs}(c) \cup \mathsf{seqs}(d))) \\
= \;&\mathsf{sub}(\mathsf{mt}(\mathsf{seqs}(c))) \cup \mathsf{sub}(\mathsf{mt}(\mathsf{seqs}(d))) && \text{(distributivity; Lemma 3.18)} \\
= \;&\mathsf{sub}(\mathsf{mt}(c)) \cup \mathsf{sub}(\mathsf{mt}(d)) && \text{(Corollary 3.13)}
\end{aligned}$$

□

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

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

Fig. 15. Negation normal form

To handle negation, we translate predicates into a *negation normal form* where only primitive predicates α can be negated (Figure 15). The translation nnf uses De Morgan's laws to push negations inwards. These possibly negated predicates are commonly called "atoms". In our setting, it is important that negation normal form is monotonic in the maximal subterm ordering ($\preceq$).

Lemma 3.21 (Negation normal form is monotonic). *If $a \preceq b$ then $\mathsf{nnf}(\neg a) \preceq \neg b$.*

Proof. By induction on a.

($a = 0$) We have $\mathsf{nnf}(\neg 0) = 1$ and $1 \preceq \neg b$ by definition.

($a = 1$) We have $\mathsf{nnf}(\neg 1) = 0$ and $0 \preceq \neg b$ by definition.

($a = \alpha$) We have $\mathsf{nnf}(\neg \alpha) = \neg \alpha$; since $a \preceq b$, it must be that $\alpha \in \mathsf{sub}(\mathsf{mt}(b))$, so $\neg \alpha \in \mathsf{sub}(\mathsf{mt}(\neg b))$. We have $\alpha \in \mathsf{sub}(\neg b)$, since $\alpha \in \mathsf{sub}(b)$.

($a = \neg c$) We have $\mathsf{nnf}(\neg\neg c) = \mathsf{nnf}(c)$; since $c \in \mathsf{sub}(a)$ and $a \preceq b$, it must be that $c \in \mathsf{sub}(\mathsf{mt}(b))$, so $\mathsf{nnf}(c) \preceq \neg b$ by the IH.

($a = c + d$) We have $\mathsf{nnf}(\neg(c + d)) = \mathsf{nnf}(\neg c) \cdot \mathsf{nnf}(\neg d)$; since c and d are subterms of a and $a \preceq b$, $\neg c$ and $\neg d$ must be in $\mathsf{sub}(\mathsf{mt}(\neg b))$, and we are done by the IHs.

($a = c \cdot d$) We have $\mathsf{nnf}(\neg(c \cdot d)) = \mathsf{nnf}(\neg c) + \mathsf{nnf}(\neg d)$; since c and d are subterms of a and $a \preceq b$, $\neg c$ and $\neg d$ must be in $\mathsf{sub}(\mathsf{mt}(\neg b))$, and we are done by the IHs.

□

Lemma 3.22 (Normal form ordering). *For all tests a, b, c and normal forms x, y, z, the following inequalities hold:*

(1) $a \preceq a \cdot b$ (extension);
(2) if $a \in \mathsf{tests}(x)$, then $a \preceq x$ (subsumption);
(3) $x \approx \sum_{a \in \mathsf{tests}(x)} a$ (equivalence);
(4) if $x \preceq x'$ and $y \preceq y'$, then $x + y \preceq x' + y'$ (normal-form parallel congruence);
(5) if $x + y \preceq z$, then $x \preceq z$ and $y \preceq z$ (inversion);
(6) if $a \preceq a'$ and $b \preceq b'$, then $a \cdot b \preceq a' \cdot b'$ (test sequence congruence);
(7) if $a \preceq x$ and $b \preceq x$ then $a \cdot b \preceq x$ (test bounding);
(8) if $a \preceq b$ and $x \preceq c$ then $a \cdot x \preceq b \cdot c$ (mixed sequence congruence);
(9) if $a \preceq b$ then $\mathsf{nnf}(\neg a) \preceq \neg b$ (negation normal-form monotonic).

Each of the above equalities also hold replacing $\preceq$ with $\prec$, excluding the equivalence (3).

Proof. We prove each properly independently and in turn. Each property can be proved using the foregoing lemmas and set-theoretic reasoning. □

Lemma 3.23 (Test sequence split). *If $a \in \mathrm{mt}(c)$ then $c \equiv a \cdot b$ for some $b \prec c$.*

Proof. We have $a \in \mathrm{seqs}(c)$ by definition. Suppose $\mathrm{seqs}(c) = \{a, c_1, \dots, c_k\}$. By sequence extraction, we have $c \equiv a \cdot c_1 \cdot \dots \cdot c_k$ (Lemma 3.12). So let $b = c_1 \cdot \dots \cdot c_k$; we must show $b \prec c$, i.e., $\mathrm{sub}(\mathrm{mt}(b)) \subsetneq \mathrm{sub}(\mathrm{mt}(c))$. Note that $\{c_1, \dots, c_k\} = \mathrm{seqs}(b)$. We find:

$$
\begin{array}{rcll}
\mathrm{sub}(\mathrm{mt}(b)) & \subsetneq & \mathrm{sub}(\mathrm{mt}(c)) & \\
& \Updownarrow & & \text{(Corollary 3.13)} \\
\mathrm{sub}(\mathrm{mt}(\mathrm{seqs}(b))) & \subsetneq & \mathrm{sub}(\mathrm{mt}(\mathrm{seqs}(c))) & \\
& \Updownarrow & & \\
\mathrm{sub}(\mathrm{mt}(\{c_1, \dots, c_k\})) & \subsetneq & \mathrm{sub}(\mathrm{mt}(\{a, c_1, \dots, c_k\})) & \\
& \Updownarrow & & \text{(distributivity; Lemma 3.18)} \\
\bigcup_{i=1}^{k} \mathrm{sub}(\mathrm{mt}(\{c_i\})) & \subsetneq & \mathrm{sub}(\mathrm{mt}(a)) \cup \bigcup_{i=1}^{k} \mathrm{sub}(\mathrm{mt}(\{c_i\})) &
\end{array}
$$

Since $a \in \mathrm{mt}(c)$, we know that $a \notin \mathrm{sub}(\mathrm{mt}(c_i))$ for all i. But terms are subterms of themselves (Lemma 3.6), so $a \in \mathrm{sub}(a) = \mathrm{sub}(\mathrm{mt}(a))$. □

Lemma 3.24 (Maximal test inequality). *If $a \in \mathrm{mt}(y)$ and $x \preceq y$ then either $a \in \mathrm{mt}(x)$ or $x \prec y$.*

Proof. Since $a \in \mathrm{mt}(y)$, we have $a \in \mathrm{sub}(\mathrm{mt}(y))$. Since $x \preceq y$, we know that $\mathrm{sub}(\mathrm{mt}(x)) \subseteq \mathrm{sub}(\mathrm{mt}(y))$. We go by cases on whether or not $a \in \mathrm{mt}(x)$:

($a \in \mathrm{mt}(x)$) We are done immediately.

($a \notin \mathrm{mt}(x)$) In this case, we show that $a \notin \mathrm{sub}(\mathrm{mt}(x))$ and therefore $x \prec y$. Suppose, for a contradiction, that $a \in \mathrm{sub}(\mathrm{mt}(x))$. Since $a \notin \mathrm{mt}(x)$, there must exist some $b \in \mathrm{sub}(\mathrm{mt}(x))$ where $a \in \mathrm{sub}(b)$. But since $x \preceq y$, we must also have $b \in \mathrm{sub}(\mathrm{mt}(y))$... and so it couldn't be that case that $a \in \mathrm{mt}(y)$). We can conclude that it must, then, be the case that $a \notin \mathrm{sub}(\mathrm{mt}(x))$ and so $x \prec y$. □

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

Lemma 3.25 (Splitting). *If $a \in \mathrm{mt}(x)$, then there exist y and z such that $x \equiv a \cdot y + z$ and $y \prec x$ and $z \prec x$.*

Proof. Suppose $x = \sum_{i=1}^{k} c_i \cdot m_i$. We have $a \in \mathrm{mt}(x)$, so, in particular:

$$a \in \mathrm{seqs}(\mathrm{tests}(x)) = \mathrm{seqs}(\mathrm{tests}(\sum_{i=1}^{k} c_i \cdot m_i)) = \mathrm{seqs}(\{c_1, \dots, c_k\}) = \bigcup_{i=1}^{k} \mathrm{seqs}(c_i).$$

That is, $a \in \mathrm{seqs}(c_i)$ for at least one i. We can, without loss of generality, rearrange x into two sums, where the first j elements have a in them but the rest don't, i.e., $x \equiv \sum_{i=1}^{j} c_i \cdot m_i + \sum_{i=j+1}^{k} c_i \cdot m_i$ where $a \in \mathrm{seqs}(c_i)$ for $1 \le i \le j$ but $a \notin \mathrm{seqs}(c_i)$ for $j+1 \le i \le k$. By subsumption (Lemma 3.22), we have $c_i \preceq x$. Since $a \in \mathrm{mt}(x)$, it must be that $a \in \mathrm{mt}(c_i)$ for $1 \le i \le j$ (instantiating Lemma 3.24 with the normal form $c_i \cdot 1$). By test sequence splitting (Lemma 3.23), we find that $c_i \equiv a \cdot b_i$ with $b_i \prec c_i \preceq x$ for $1 \le i \le j$, as well.

We are finally ready to produce y and z: they are the first j tests with a removed and the remaining tests which never had a, respectively. Formally, let $y = \sum_{i=1}^{j} b_i \cdot m_i$; we immediately have that $a \cdot y \equiv \sum_{i=1}^{j} c_i \cdot m_i$; let $z = \sum_{i=j+1}^{k} c_i \cdot m_i$. We can conclude that $x \equiv a \cdot y + z$.

It remains to be seen that $y \prec x$ and $z \prec x$. The argument is the same for both; presenting it for y, we have $a \notin \mathrm{seqs}(y)$ (because of sequence splitting), so $a \notin \mathrm{sub}(\mathrm{mt}(y))$. But we assumed

$a \in \mathrm{mt}(x)$, so $a \in \mathrm{sub}(\mathrm{mt}(x))$, and therefore $y \prec x$. The argument for z is nearly identical but needs no recourse to sequence splitting—we never had any $a \in \mathrm{seqs}(c_i)$ for $j+1 \le i \le k$. □

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

3.3.2 Pushback. In order to define normalization—necessary for completeness (Sec. 3.4)—the client theory must have a *weakest preconditions* operation that respects the subterm ordering.

Definition 3.26 (Weakest preconditions). The *weakest precondition* operation of the client theory is a relation $\mathrm{WP} \subseteq \mathcal{T}_\pi \times \mathcal{T}_\alpha \times \mathcal{P}(\mathcal{T}^*_{\mathrm{pred}})$, where $\mathcal{T}_\pi$ are the primitive actions and $\mathcal{T}_\alpha$ are the primitive predicates of $\mathcal{T}$. We write the relation as $\pi \cdot \alpha \ \mathrm{WP} \ \sum a_i \cdot \pi$ and read it as "α pushes back through π to yield $\sum a_i \cdot \pi$"; the second π is redundant but written for clarity. We require that if $\pi \cdot \alpha \ \mathrm{WP} \ \{a_1, \ldots, a_k\} \cdot \pi$, then $\pi \cdot \alpha \equiv \sum_{i=1}^{k} a_i \cdot \pi$, and $a_i \preceq \alpha$.

Given the client theory's weakest-precondition relation WP, we define a normalization procedure for $\mathcal{T}^*$ by extending the client's WP relation to a more general *pushback* relation, PB (Fig. 16). The client's WP relation need not be a function, nor do the a_i need to be obviously related to α or π in any way. Even when the WP relation is a function, the PB relation will generally not be a function. While WP computes the classical weakest precondition, the PB relations do something different: when pushing back we have the freedom to *change the program itself*—not normally an option for weakest preconditions (see Sec. 6).

We define the top-level normalization routine with the $p \ \mathrm{norm} \ x$ relation (Fig. 16), a syntax directed relation that takes a term p and produces a normal form $x = \sum_i a_i m_i$. Most syntactic forms are easy to normalize: predicates are already normal forms (Pred); primitive actions π are normal forms where there's just one summand and the predicate is 1 (Act); and parallel composition of two normal forms means just joining the sums (Par). But sequence and Kleene star are harder: we define judgments using PB to lift these operations to normal forms (Seq, Star).

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

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. 16), writing $x^* \ \mathrm{PB}^* \ y$ to mean that the Kleene star of x is equivalent to the normal form y—the * on the left is again suggestive notation. The PB^* relation is more subtle than PB^{J}. There are four possible ways to treat x, based on how it splits (Lemma 3.25): if $x = 0$, then our work is trivial since $0^* \equiv 1$ (StarZero); if x splits into $a \cdot x'$ where a is a maximal test and there are no other summands, then we can either use the KAT sliding lemma (Lemma 3.29)to pull the test out when a is strictly the largest test in x (Slide) or by using the KAT expansion lemma (Lemma 3.32) otherwise (Expand); if x splits into $a \cdot x' + z$, we use the KAT denesting lemma (Lemma 3.30)to pull a out before recurring on what remains (Denest).

The bulk of the pushback's work happens in the $\mathrm{PB}^\bullet$ relation, which pushes a test back through a restricted action; PB^{R} and PB^{T} use $\mathrm{PB}^\bullet$ to push tests back through normal forms and normal forms back through restricted actions, respectively. We write $m \cdot a \ \mathrm{PB}^\bullet \ y$ to mean that pushing the test a back through restricted action m yields the equivalent normal form y. The $\mathrm{PB}^\bullet$ relation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 16. Normalization for $\mathcal{T}^*$

works by analyzing both the action and the test. The client theory's WP relation is used in $\text{PB}^{\bullet}$ when we try to push a primitive predicate α through a primitive action π (Prim); all other KAT predicates can be handled by rules matching on the action or predicate structure, deferring to other PB relations. To handle negation, the function nnf translates predicates to *negation normal form*, where negations only appear on primitive predicates (Fig. 15); Pushback-Neg justifies the PrimNeg case(Pushback-Neg); we use nnf to respect the maximal subterm ordering.

Definition 3.27 (Negation normal form). The *negation normal form* of a term p is a term p' such that $p \equiv p'$ and negations occur only on primitive predicates in p'.

Lemma 3.28 (Terms are equivalent to their negation-normal forms). $\mathsf{nnf}(p) \equiv p$ *and* $\mathsf{nnf}(p)$ *is in negation normal form.*

Proof. By induction on the size of p. The only interesting case is when $p = \neg a$; we go by cases on a.

($a = 0$) We have $\neg 0 \equiv 1$ immediately, and the latter is clearly negation free.

($a = 1$) We have $\neg 1 \equiv 0$; as above.

($a = \alpha$) We have $\neg alpha$, which is in normal form.

($a = b + c$) We have $\neg(b + c) \equiv \neg b \cdot \neg c$ as a consequence of BA-Excl-Mid and soundness (Theorem 3.5). By the IH on $\neg b$ and $\neg c$, we find that $\mathsf{nnf}(\neg b) \equiv \neg b$ and $\mathsf{nnf}(\neg c) \equiv \neg c$—where the left-hand sides are negation normal. So transitively, we have $\neg(b + c) \equiv \mathsf{nnf}(\neg b) \cdot \mathsf{nnf}(\neg c)$, and the latter is negation normal.

($a = b \cdot c$) We have $\neg(b \cdot c) \equiv \neg b + \neg c$ as a consequence of BA-Excl-Mid and soundness (Theorem 3.5). By the IH on $\neg b$ and $\neg c$, we find that $\mathsf{nnf}(\neg b) \equiv \neg b$ and $\mathsf{nnf}(\neg c) \equiv \neg c$—where the left-hand sides are negation normal. So transitively, we have $\neg(b \cdot c) \equiv \mathsf{nnf}(\neg b) + \mathsf{nnf}(\neg c)$, and the latter is negation normal. □

To elucidate the way PB$^\bullet$ handles structure, suppose we have the term $(\pi_1 + \pi_2) \cdot (\alpha_1 + \alpha_2)$. One of two rules could apply: we could split up the tests and push them through individually (SeqParTest), or we could split up the actions and push the tests through together (SeqParAction). It doesn't particularly matter which we do first: the next step will almost certainly be the other rule, and in any case the results will be equivalent from the perspective of our equational theory. It *could* be the case that choosing a one rule over another could give us a smaller term, which might yield a more efficient normalization procedure. Similarly, a given normal form may have more than one maximal test—and therefore be splittable in more than one way (Lemma 3.25)—and it may be that different splits produce more or less efficient terms. We haven't yet studied differing strategies for pushback.

Lemma 3.29 (Sliding). $p \cdot (q \cdot p)^* \equiv (p \cdot q)^* \cdot p$.

Proof. Following Kozen [35], as a corollary of a related result: if $p \cdot x \equiv x \cdot q$ then $p^* \cdot x \equiv x \cdot q^*$. We prove this separate property by mutual inclusion.

($\Rightarrow$) We use KA-LFP-L with $p = p$ and $q = x$ and $r = x \cdot q^*$. We must show that $x + p \cdot x \cdot q^* \leq x \cdot q^*$ to find $p^* \cdot x \leq x \cdot q^*$.

If $p \cdot q \leq x \cdot q$ then $p \cdot x \cdot q^* \leq x \cdot q \cdot q^*$ by monotonicity. We have $x + x \cdot q \cdot q^* \leq x \cdot q^*$ by KA-Unroll-L and KA-Plus-Idem. Therefore $x + p \cdot x \cdot q^* \leq x + x \cdot q \cdot q^* \leq x \cdot q^*$, as desired.

($\Leftarrow$) This case is symmetric to the first, using -R rules instead of -L rules. We apply KA-LFP-R with $p = x$ and $r = q$ and $q = p^* \cdot x$. We must show $x + p^* \cdot x \cdot q \leq p^* \cdot x$ to find $x \cdot q^* \leq p^* \cdot x$.

If $x \cdot q \leq p \cdot x$, then $p^* \cdot x \cdot q \leq p^* \cdot p \cdot x$ by monotonicity. We have $x + p^* \cdot p \cdot x \leq p^* \cdot x$ by KA-Unroll-R and KA-Plus-Idem. Therefore $x + p^* \cdot x \cdot q \leq x + p^* \cdot p \cdot x \leq p^* \cdot x$, as desired.

We can now find sliding by letting $p = p \cdot q$ and $x = p$ and $q = q \cdot p$ in the above, i.e., we have the premise $p \cdot q \cdot p \equiv p \cdot q \cdot p$ by reflexivity, and so $(p \cdot q)^* \cdot p \equiv p \cdot (q \cdot p)^*$. □

Lemma 3.30 (Denesting). $(p + q)^* \equiv p^* \cdot (q \cdot p^*)^*$.

Proof. Following Kozen [35], we do the proof by mutual inclusion. The proof is surprisingly challenging, so we include it here.

($\Rightarrow$) To show $(p+q)^* \leq a^* \cdot (b \cdot a^*)^*$, we apply induction with $q = 1$ and $r = p^* \cdot (q \cdot p^*)^*$ (to show $(a+b)^* \cdot 1 \leq r$). We must show that $1 + (p+q) \cdot p^* \cdot (q \cdot p^*)^* \leq p^* \cdot (q \cdot p^*)^*$. We do so in several parts, working our way there in five steps.

First, we observe that $1 \leq p^* \cdot (q \cdot p^*)^*$ (A) because:

$$\begin{array}{rll} & 1 + p^* \cdot (q \cdot p^*)^* & \\ \equiv & 1 + (1 + p \cdot p^* \cdot (q \cdot p^*)^*) & \text{KA-Unroll-L} \\ \equiv & 1 + p \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Assoc, KA-Plus-Idem} \\ \equiv & p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L} \end{array}$$

Next, $p \cdot p^* \cdot (q \cdot p^*)^* \leq p^* \cdot (q \cdot p^*)^*$ (B) because:

$$\begin{array}{rll} & p \cdot p^* \cdot (q \cdot p^*)^* + p^* \cdot (q \cdot p^*)^* & \\ \equiv & p \cdot p^* \cdot (q \cdot p^*)^* + 1 + p \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L} \\ \equiv & 1 + p \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Idem} \\ \equiv & p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L} \end{array}$$

We have $q \cdot p^* \cdot (q \cdot p^*)^* \leq (q \cdot p^*)^*$ because:

$$\begin{array}{rll} & q \cdot p^* \cdot (q \cdot p^*)^* + (q \cdot p^*)^* & \\ \equiv & q \cdot p^* \cdot (q \cdot p^*)^* + 1 + q \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L} \\ \equiv & 1 + q \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Idem} \\ \equiv & (q \cdot p^*)^* & \text{KA-Unroll-L} \end{array}$$

Further, $(q \cdot p^*)^* \leq p^* \cdot (q \cdot p^*)^*$ because:

$$\begin{array}{rll} & (q \cdot p^*)^* + p^* \cdot (q \cdot p^*)^* & \\ \equiv & (q \cdot p^*)^* + 1 \cdot (q \cdot p^*)^* + p \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L, KA-Dist-R} \\ \equiv & 1 \cdot (q \cdot p^*)^* + p \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Idem} \\ \equiv & p^* \cdot (q \cdot p^*)^* & \text{KA-Unroll-L} \end{array}$$

Finally, $q \cdot p^* \cdot (q \cdot p^*)^* \leq a^* \cdot (q \cdot p^*)^*$ (C) by transitivity with the last two results.

Now we can find that

$$1 + (p+q)p^* \cdot (q \cdot p^*)^* \leq 1 + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* \leq p^* \cdot (q \cdot p^*)^*$$

because:

$$\begin{array}{rll} & 1 + (p+q)p^* \cdot (q \cdot p^*)^* + 1 + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* & \\ \equiv & 1 + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Idem} \\ \equiv & 1 + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* & \text{KA-Plus-Idem} \end{array}$$

because, finally:

$$\begin{array}{rll} & 1 + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* + p^* \cdot (q \cdot p^*)^* & \\ \equiv & p^* \cdot (q \cdot p^*)^* + p \cdot p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* & \text{(A)} \\ \equiv & p^* \cdot (q \cdot p^*)^* + q \cdot p^* \cdot (q \cdot p^*)^* & \text{(B)} \\ \equiv & p^* \cdot (q \cdot p^*)^* & \text{(C)} \end{array}$$

($\Leftarrow$) To show $p^* \cdot (q \cdot p^*)^* \leq (p+q)^*((p+q)(p+q)^*)^*$, we have first that $p \leq p+q$ and $q \leq p+q$, and so $p+q \leq (p+q)^*$. And so, by monotonicity $p^* \cdot (q \cdot p^*)^* \leq (p+q)^*((p+q)(p+q)^*)^*$. We can then find that $(p+q)^* \cdot ((p+q) \cdot (p+q)^*)^* \leq (p+q)^* \cdot ((p+q)^*)^*$ because:

$$
\begin{array}{lll}
 & (p+q)(p+q)^* + (p+q)^* & \\
\equiv & p \cdot (p+q)^* + q \cdot (p+q)^* + (p+q)^* & \text{KA-Dist-R} \\
\equiv & p \cdot (p+q)^* + q \cdot (p+q)^* + 1 + (p+q)(p+q)^* & \text{KA-Unroll-L} \\
\equiv & p \cdot (p+q)^* + q \cdot (p+q)^* + 1 + p \cdot (p+q)^* + q \cdot (p+q)^* & \text{KA-Dist-R} \\
\equiv & 1 + p \cdot (p+q)^* + q \cdot (p+q)^* & \text{KA-Plus-Idem} \\
\equiv & 1 + (p+q) \cdot (p+q)^* & \text{KA-Dist-R} \\
\equiv & (p+q)^* & \text{KA-Unroll-L}
\end{array}
$$

But we also have $(p+q)^* \cdot ((p+q)^*)^* \leq (p+q)^*$ because:

$$
\begin{array}{lll}
 & (p+q)^* \cdot ((p+q)^*)^* + (p+q)^* & \\
\equiv & (p+q)^* \cdot (p+q)^* + (p+q)^* & \text{because } (x^*)^* = x^* \\
\equiv & (p+q)^* + (p+q)^* & \text{because } x^*x^* = x^* \\
\equiv & (p+q)^* & \text{KA-Plus-Idem}
\end{array}
$$

□

Lemma 3.31 (Star invariant). *If $p \cdot a \equiv a \cdot q + r$ then $p^* \cdot a \equiv (a + p^* \cdot r) \cdot q^*$.*

Proof. We show two implications using $\leq$ to derive the equality.

($\Rightarrow$) We want to show $p^*; a \leq (a + p^*; y); x^*$.

We know that $q + pr \leq r \implies p^*q \leq r$ by the induction axiom KA-LFP-L, so we can instantiate it with p as p and q as a and r as $(a + p^*; y); x^*$. We find:

$$
\begin{array}{rcl}
a + p; (a + p^*; y); x^* & \leq & (a + p^*; y); x^* \\
a + p; a; x^* + p; p^*; y; x^* & \leq & (a + p^*; y); x^* \\
a + p; a; x^* + p; p^*; y; x^* + (a + p^*; y); x^* & = & (a + p^*; y); x^* \\
a + p; a; x^* + p; p^*; y; x^* + a; x^* + p^*; y; x^* & = & (a + p^*; y); x^* \\
(a + a; x^* + p; a; x^*) + (p; p^*; y; x^* + p^*; y; x^*) & = & (a + p^*; y); x^* \\
(a; x^* + p; a; x^*) + (p; p^*; y; x^* + p^*; y; x^*) & = & (a + p^*; y); x^* \\
(1 + p); a; x^* + (1 + p); p^*; y; x^* & = & (a + p^*; y); x^* \\
a; x^* + p^*; y; x^* & = & (a + p^*; y); x^* \\
(a + p^*; y); x^* & = & (a + p^*; y); x^*
\end{array}
$$

($\Leftarrow$) We can to show $(a + p^*; y); x^* \leq p^*; a$ We can apply the other induction axiom (KA-LFP-R), $q + r; p \leq r \implies q; p^* \leq r$, with $p = x$ and $q = (a + p^*; y)$ and $r = p^*; a$. We find:

$$
\begin{array}{rcl}
(a + p^*; y) + (p^*; a); x & \leq & p^*; a \\
a + p^*; y + p^*; a; x + p^*; a & = & p^*; a \\
a + p^*; (a; x + y + a) & = & p^*; a \\
a + p^*; (p; a + a) & = & p^*; a \\
a + p^*; (a; (p + 1)) & = & p^*; a \\
a + p^*; a & = & p^*; a \\
p^*; a & = & p^*; a
\end{array}
$$

□

Lemma 3.32 (Star expansion). *If $p \cdot a \equiv a \cdot q + r$ then $p \cdot a \cdot (p \cdot a)^* \equiv (a \cdot q + r) \cdot (q + r)^*$.*

Proof. First we observe that $p; a; (p; a)^*$ is equivalent to $(p; a)^*; p; a$ (apply KA-SLIDING twice). We show two implications using $\leq$ to derive the equality.

($\Rightarrow$) We want to show $(p; a)^*; p; a \leq (a; x + y); (x + y)^*$.

We know that $q + pr \leq r \implies p^*q \leq r$ by the induction axiom KA-LFP-L, so we can instantiate it with p and q as $p; a$ and r as $(a; x + y); (x + y)^*$. We find:

$$\begin{array}{rcl}
p; a + p; a; (a; x + y); (x + y)^* & \leq & (a; x + y); (x + y)^* \\
p; a + (p; a; x + p; a; y); (x + y)^* & \leq & (a; x + y); (x + y)^* \\
(a; x + y) + ((a; x + y); x + (a; x + y); y); (x + y)^* & \leq & (a; x + y); (x + y)^* \\
(a; x + y) + (a; x + y); (x + y); (x + y)^* & \leq & (a; x + y); (x + y)^* \\
(a; x + y); (1 + (x + y); (x + y)^*) & \leq & (a; x + y); (x + y)^* \\
(a; x + y); (x + y)^* & \leq & (a; x + y); (x + y)^*
\end{array}$$

($\Leftarrow$)) We can to show $(a; x + y); (x + y)^* \leq p; a; (p; a)^*$ We can apply the other induction axiom (KA-LFP-R), $q + r; p \leq r \implies q; p^* \leq r$, with $p = x + y$ and $q = a; x + y$ and $r = p; a(p; a)^*$. We find:

$$\begin{array}{rcl}
(a; x + y) + p; a; (p; a)^*; (x + y) & \leq & (p; a)^*; p; a \\
p; a + p; a; (p; a)^*; (x + y) & \leq & (p; a)^*; p; a \\
p; a + p; a; (p; a)^*; (x + y) & \leq & p; a + (p; a)^*; p; a; p; a \\
p; a + p; a; (p; a)^*; (x + y) & \leq & p; a + (p; a)^*; p; a; (a; x + y) \\
p; a + p; a; (p; a)^*; (x + y) & \leq & p; a + (p; a)^*; (a; x + y); (x + y) \\
p; a + p; a; (p; a)^*; (x + y) & \leq & p; a + (p; a)^*; (p; a); (x + y)
\end{array}$$

□

Lemma 3.33 (Pushback through primitive actions). *Pushing a test back through a primitive action leaves the primitive action intact, i.e., if* $\pi \cdot a \ \mathsf{PB}^\bullet \ x$ *or* $(\sum b_i \cdot \pi) \cdot a \ \mathsf{PB}^\mathsf{T} \ x$, *then* $x = \sum a_i \cdot \pi$.

Proof. By induction on the derivation rule used. □

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 3.34). Once we've established that our pushback relations' derivations mean what we want, we have to show that we can find such derivations; here we use our maximal subterm measure to show that the recursive tangle of our PB relations always terminates (Theorem 3.35) , which makes extensive use of our subterm ordering lemma (Lemma 3.22) and splitting (Lemma 3.25)).

Theorem 3.34 (Pushback soundness).

(1) *If* $x \cdot y \ \mathsf{PB}^\mathsf{J} \ z'$ *then* $x \cdot y \equiv z'$.
(2) *If* $x^* \ \mathsf{PB}^* \ y$ *then* $x^* \equiv y$.
(3) *If* $m \cdot a \ \mathsf{PB}^\bullet \ y$ *then* $m \cdot a \equiv y$.
(4) *If* $m \cdot x \ \mathsf{PB}^\mathsf{R} \ y$ *then* $m \cdot x \equiv y$.
(5) *If* $x \cdot a \ \mathsf{PB}^\mathsf{T} \ y$ *then* $x \cdot a \equiv y$.

Proof. By simultaneous induction on the derivations. Cases are grouped by judgment.

Sequential composition of normal forms ($x \cdot y$ PB^{J} z).

(Join) We have $x = \sum_{i=1}^{k} a_i \cdot m_i$ and $y = \sum_{j=1}^{l} b_j \cdot n_j$. By the IH on (3), each $m_i \cdot b_j$ $\mathsf{PB}^{\bullet}$ x_{ij}. We compute:

$$
\begin{array}{rll}
 & x \cdot y & \\
\equiv & \left[\sum_{i=1}^{k} a_i \cdot m_i\right] \cdot \left[\sum_{j=1}^{l} b_j \cdot n_j\right] & \\
\equiv & \sum_{i=1}^{k} a_i \cdot m_i \cdot \left[\sum_{j=1}^{l} b_j \cdot n_j\right] & \text{(KA-Dist-R)} \\
\equiv & \sum_{i=1}^{k} a_i \cdot \left[m_i \cdot \sum_{j=1}^{l} b_j \cdot n_j\right] & \text{(KA-Seq-Assoc)} \\
\equiv & \sum_{i=1}^{k} a_i \cdot \left[\sum_{j=1}^{l} m_i \cdot b_j \cdot n_j\right] & \text{(KA-Dist-L)} \\
\equiv & \sum_{i=1}^{k} a_i \cdot \left[\sum_{j=1}^{l} x_{ij} \cdot n_j\right] & \text{(IH (3))} \\
\equiv & \sum_{i=1}^{k} \sum_{j=1}^{l} a_i \cdot x_{ij} \cdot n_j & \text{(KA-Dist-L)}
\end{array}
$$

Kleene star of normal forms (x^* PB^{J} y).

(StarZero) We have 0^* PB^* 1. We compute:

$$
\begin{array}{rll}
 & 0^* & \\
\equiv & 1 + 0 \cdot 0^* & \text{(KA-Unroll-L)} \\
\equiv & 1 + 0 & \text{(KA-Zero-Seq)} \\
\equiv & 1 & \text{(KA-Plus-Zero)}
\end{array}
$$

(Slide) We are trying to pushback the minimal term a of x through a star, i.e., we have $(a \cdot x)^*$; by the IH on (5), we know there exists some y such that $x \cdot a \equiv y$; by the IH on (2), we know that $y^* \equiv y'$; and by the IH on (1), we know that $y' \cdot x \equiv z$. We must show that $(a \cdot x)^* \equiv 1 + a \cdot z$. We compute:

$$
\begin{array}{rll}
 & (a \cdot x)^* & \\
\equiv & 1 + a \cdot x \cdot (a \cdot x)^* & \text{(KA-Unroll-L)} \\
\equiv & 1 + a \cdot (x \cdot a)^* \cdot x & \text{(sliding with } p = x \text{ and } q = a\text{; Lemma 3.29)} \\
\equiv & 1 + a \cdot y^* \cdot x & \text{(IH (5))} \\
\equiv & 1 + a \cdot y' \cdot x & \text{(IH (2))} \\
\equiv & 1 + a \cdot z & \text{(IH (1))}
\end{array}
$$

(Expand) We are trying to pushback the minimal term a of x through a star, i.e., we have $(a \cdot x)^*$; by the IH on (5), we know that there exist t and u such that $x \cdot a \equiv a \cdot t + u$; by the IH on (2), we know that there exists a y such that $(t + u)^* \equiv y$; and by the IH on (1), we know that there is some z such that $y \cdot x \equiv z$. We compute:

$$
\begin{array}{cll}
& (a \cdot x)^* & \\
\equiv & 1 + a \cdot x + a \cdot x \cdot a \cdot x \cdot (a \cdot x)^* & (\textsc{KA-Unroll-L}) \\
\equiv & 1 + a \cdot x + a \cdot x \cdot a \cdot (x \cdot a)^* \cdot x & \\
& & (\text{sliding with } p = x \text{ and } q = a; \text{Lemma 3.29}) \\
\equiv & 1 + a \cdot x + a \cdot [x \cdot a \cdot (x \cdot a)^*] \cdot x & (\textsc{KA-Seq-Assoc}) \\
\equiv & 1 + a \cdot x + a \cdot [(a \cdot t + u) \cdot (t + u)^*] \cdot x & \\
& & (\text{expansion using IH (5); Lemma 3.32}) \\
\equiv & 1 + a \cdot x + a \cdot (a \cdot t + u) \cdot (t + u)^* \cdot x & (\textsc{KA-Seq-Assoc}) \\
\equiv & 1 + a \cdot x + (a \cdot a \cdot t + a \cdot u) \cdot (t + u)^* \cdot x & (\textsc{KA-Dist-L}) \\
\equiv & 1 + a \cdot x + (a \cdot t + a \cdot u) \cdot (t + u)^* \cdot x & (\textsc{BA-Seq-Idem}) \\
\equiv & 1 + a \cdot x + a \cdot (t + u) \cdot (t + u)^* \cdot x & (\textsc{BA-Seq-Idem}) \\
\equiv & 1 + a \cdot 1 \cdot x + a \cdot (t + u) \cdot (t + u)^* \cdot x & (\textsc{KA-One-Seq}) \\
\equiv & 1 + (a \cdot 1 + a \cdot (t + u) \cdot (t + u)^*) \cdot x & (\textsc{KA-Dist-R}) \\
\equiv & 1 + a \cdot (1 + (t + u) \cdot (t + u)^*) \cdot x & (\textsc{KA-Dist-L}) \\
\equiv & 1 + a \cdot (t + u)^* \cdot x & (\textsc{KA-Unroll-L}) \\
\equiv & 1 + a \cdot y \cdot x & (\text{IH (2)}) \\
\equiv & 1 + a \cdot z & (\text{IH (1)})
\end{array}
$$

(Denest) We have a compound normal form $a \cdot x + y$ under a star; we will push back the maximal test a. By our first IH on (2) we know that that $y^* \equiv y'$ for some y'; by our first IH on (1), we know that $x \cdot y' \equiv x'$ for some x'; by our second IH on (2), we know that $(a \cdot x')^* \equiv z$ for some z; and by our second IH on (1), we know that $y' \cdot z \equiv z'$ for some z'. We must show that $(a \cdot x + y)^* \equiv z'$. We compute:

$$
\begin{array}{cll}
& (a \cdot x + y)^* & \\
\equiv & y^* \cdot (a \cdot x \cdot y^*)^* & (\text{denesting with } p = a \cdot x \text{ and } q = y; \text{Lemma 3.30}) \\
\equiv & y' \cdot (a \cdot x \cdot y')^* & (\text{first IH (2)}) \\
\equiv & y' \cdot (a \cdot x')^* & (\text{first IH (1)}) \\
\equiv & y' \cdot z & (\text{second IH (2)}) \\
\equiv & z' & (\text{second IH (1)})
\end{array}
$$

Pushing tests through actions ($m \cdot a$ PB$^\bullet$ y).

(SeqZero) We are pushing 0 back through a restricted action m. We immediately find $m \cdot 0 \equiv 0$ by KA-Seq-Zero.

(SeqOne) We are pushing 1 back through a restricted action m. We find:

$$
\begin{array}{cll}
& m \cdot 1 & \\
\equiv & m & (\textsc{KA-One-Seq}) \\
\equiv & 1 \cdot m & (\textsc{KA-Seq-One})
\end{array}
$$

(SeqSeqTest) We are pushing the tests $a \cdot b$ through the restricted action m. By our first IH on (3), we have $m \cdot a \equiv y$; by our second IH on (3), we have $y \cdot b \equiv z$. We compute:

$$
\begin{array}{cll}
& m \cdot (a \cdot b) & \\
\equiv & m \cdot a \cdot b & (\textsc{KA-Seq-Assoc}) \\
\equiv & y \cdot b & (\text{first IH (3)}) \\
\equiv & z & (\text{second IH (3)})
\end{array}
$$

(SeqSeqAction) We are pushing the test a through the restricted actions $m \cdot n$. By our IH on (3), we have $n \cdot a \equiv x$; by our IH on (4), we have $m \cdot x \equiv y$. We compute:

$$\begin{array}{rll} & (m \cdot n) \cdot a & \\ \equiv & m \cdot (n \cdot a) & (\text{KA-Seq-Assoc}) \\ \equiv & m \cdot x & (\text{IH (3)}) \\ \equiv & y & (\text{IH (4)}) \end{array}$$

(SeqParTest) We are pushing the tests $a + b$ through the restricted action m. By our first IH on (3), we have $m \cdot a \equiv x$; by our second IH on (3), we have $m \cdot b \equiv y$. We compute:

$$\begin{array}{rll} & m \cdot (a + b) & \\ & m \cdot a + m \cdot b & (\text{KA-Dist-L}) \\ \equiv & x + m \cdot b & (\text{first IH (3)}) \\ \equiv & x + y & (\text{second IH (3)}) \end{array}$$

(SeqParAction) We are pushing the test a through the restricted actions $m + n$. By our first IH on (3), we have $m \cdot a \equiv x$; by our second IH on (3), we have $n \cdot a \equiv y$. We compute:

$$\begin{array}{rll} & (m + n) \cdot a & \\ & m \cdot a + n \cdot a & (\text{KA-Dist-R}) \\ \equiv & x + n \cdot a & (\text{first IH (3)}) \\ \equiv & x + y & (\text{second IH (3)}) \end{array}$$

(Prim) We are pushing a primitive predicate α through a primitive action π. We have, by assumption, that $\pi \cdot a$ WP $\{a_1, \dots, a_k\}$. By definition of the WP relation, it must be the case that $\pi \cdot \alpha \equiv \sum_{i=1}^{k} a_i \cdot \pi$

(PrimNeg) We are pushing a negated predicate $\neg a$ back through a primitive action π. We have, by assumption, that $\pi \cdot a$ $\mathsf{PB}^\bullet$ $\sum_i a_i \cdot pi$ and that $\mathsf{nnf}(\neg(\sum_i a_i)) = b$, so $\neg(\sum_i a_i) \equiv b$ (Lemma 3.28). By the IH, we know that $\pi \cdot a \equiv \sum_i a_i \cdot \pi$; we must show that $\pi \cdot \neg a \equiv b \cdot \pi$. By our assumptions, we know that $b \cdot \pi \equiv \neg(\sum_i a_i) \cdot \pi$, so by pushback negation (Pushback-Neg/Lemma 3.1).

(SeqStarSmaller) We are pushing the test a through the restricted action m^*. By our IH on (3), we have $m \cdot a \equiv x$ for some x; by our IH on (4), we have $m^* \cdot x \equiv y$ for some y. We compute:

$$\begin{array}{rll} & m^* \cdot a & \\ \equiv & (1 + m^* \cdot m) \cdot a & (\text{KA-Unroll-R}) \\ \equiv & a + m^* \cdot m \cdot a & (\text{KA-Dist-R}) \\ \equiv & a + m^* \cdot (m \cdot a) & (\text{KA-Seq-Assoc}) \\ \equiv & a + m^* \cdot x & (\text{IH (3)}) \\ \equiv & a + y & (\text{IH (4)}) \end{array}$$

(SeqStarInv) We are pushing the test a through the restricted action m^*. By our IH on (3), there exist t and u such that $m \cdot a \equiv a \cdot t + u$; by our IH on (4), there exists an x such that $m^* \cdot u \equiv x$; by our IH on (2), there exists a y such that $u^* \equiv y$; and by our IH on (1), there exists a z such that $x \cdot y \equiv z$. We compute:

m . a = a . t + u m* a = (a + m* . u) + t*

$$\begin{array}{rlr} & m^* \cdot a & \\ \equiv & (a + m^* \cdot u) \cdot t^* & \text{(star invariant on IH (3); Lemma 3.31)} \\ \equiv & a \cdot t^* + m^* \cdot u \cdot t^* & \text{(KA-Dist-R)} \\ \equiv & a \cdot t^* + x \cdot t^* & \text{(IH (4))} \\ \equiv & a \cdot y + x \cdot y & \text{(IH (2))} \\ \equiv & a \cdot y + z & \text{(IH (1))} \end{array}$$

Pushing normal forms through actions ($m \cdot x$ PB^{R} z).

(Restricted) We have $x = \sum_{i=1}^{k} a_i \cdot n_i$. By the IH on (3), $m \cdot a_i$ $\mathsf{PB}^{\bullet}$ y_i. We compute:

$$\begin{array}{rlr} & m \cdot x & \\ \equiv & m \cdot \sum_{i=1}^{k} a_i \cdot n_i & \\ \equiv & \sum_{i=1}^{k} m \cdot a_i \cdot n_i & \text{(KA-Dist-L)} \\ \equiv & \sum_{i=1}^{k} y_i \cdot n_i & \text{(IH (3))} \end{array}$$

Pushing tests through normal forms ($x \cdot a$ PB^{T} y).

(Test) We have $x = \sum_{i=1}^{k} a_i \cdot m_i$. By the IH on (3), we have $m_i \cdot a$ $\mathsf{PB}^{\bullet}$ y_i where $y_i = \sum_{j=1}^{l} b_{ij} \cdot m_{ij}$. We compute:

$$\begin{array}{rlr} & x \cdot a & \\ \equiv & \left[\sum_{i=1}^{k} a_i \cdot m_i\right] \cdot a & \\ \equiv & \sum_{i=1}^{k} a_i \cdot m_i \cdot a & \text{(KA-Dist-R)} \\ \equiv & \sum_{i=1}^{k} a_i \cdot (m_i \cdot a) & \text{(KA-Seq-Assoc)} \\ \equiv & \sum_{i=1}^{k} a_i \cdot y_i & \text{(IH (3))} \\ \equiv & \sum_{i=1}^{k} a_i \cdot \sum_{j=1}^{l} b_{ij} \cdot m_{ij} & \\ \equiv & \sum_{i=1}^{k} \sum_{j=1}^{l} a_i \cdot b_{ij} \cdot m_{ij} & \text{(KA-Dist-L)} \end{array}$$

□

Theorem 3.35 (Pushback existence). *For all x and m and a:*

(1) For all y and z, if $x \leq z$ and $y \leq z$ then there exists some $z' \leq z$ such that $x \cdot y$ PB^{J} z'.
(2) There exists a $y \leq x$ such that x^* PB^* y.
(3) There exists some $y \leq a$ such that $m \cdot a$ $\mathsf{PB}^{\bullet}$ y.
(4) There exists a $y \leq x$ such that $m \cdot x$ PB^{R} y.
(5) If $x \leq z$ and $a \leq z$ then there exists a $y \leq z$ such that $x \cdot a$ PB^{T} y.

Proof. By induction on the lexicographical order of: the subterm ordering ($<$); the size of x(for (1), (2), (4), and (5)); the size of m(for (3) and (4)); and the size of a(for (3)).

Sequential composition of normal forms ($x \cdot y$ PB^{J} z). We have $x = \sum_{i=1}^{k} a_i \cdot m_i$ and $y = \sum_{j=1}^{l} b_j \cdot n_j$; by the IH on (3) with the size decreasing on m_i, we know that $m_i \cdot b_j$ $\mathsf{PB}^{\bullet}$ x_{ij} for each i and j such that $x_{ij} \leq a_i$, so by Join, we know that $x \cdot y$ PB^{J} $\sum_{i=1}^{k} \sum_{j=1}^{l} a_i x_{ij} n_j = z'$.

Given that $x, y \leq z$, it remains to be seen that $z' \leq z$. We've assumed that $a_i \leq x \leq z$. By our IH on (3) we found earlier that $x_{ij} \leq a_i \leq z$. Therefore, by unpacking x and applying test bounding (Lemma 3.22), $a_i \cdot x_{ij} \cdot n_j \leq z$. By normal form parallel congruence (Lemma 3.22), we have $z' \leq z$.)

Kleene star of normal forms ($x^* \mathrel{\mathsf{PB}^{\mathsf{J}}} y$). If x is vacuous, we find that $0^* \mathrel{\mathsf{PB}^*} 1$ by StarZero, with $1 \preceq 0$ since they have the same maximal terms (just 1).

If x isn't vacuous, then we have $x \equiv a \cdot x_1 + x_2$ where $x_1, x_2 \prec x$ and $a \in \mathsf{mt}(x)$ by splitting (Lemma 3.25). We first consider whether x_2 is vacuous.

(x_2 is vacuous) We have $x \equiv a \cdot x_1 + 0 \equiv a \cdot x_1$.

By our IH on (5) with x_1 decreasing in size, we have $x_1 \cdot a \mathrel{\mathsf{PB}^{\mathsf{T}}} w$ where $w \preceq x$ (because $x_1 \prec x$ and $a \preceq x$). By maximal test inequality (Lemma 3.24), we have two cases: either $a \in \mathsf{mt}(w)$ or $w \prec a \preceq x$.

($a \in \mathsf{mt}(w)$) By splitting (Lemma 3.25), we have $w \equiv a \cdot t + u$ for some normal forms $t, u \prec w$.

By normal-form parallel congruence (Lemma 3.22), $t + u \prec x$; so by the IH on (2) with our subterm ordering decreasing on $t + u \prec x$, we find that $(t + u)^* \mathrel{\mathsf{PB}^*} w'$ for some $w' \preceq (t + u)^* \prec w \preceq x$. Since $w' \prec x$, we can apply our IH on (1) with our subterm ordering decreasing on $w' \prec x$ to find that $w' \cdot x_1 \mathrel{\mathsf{PB}^{\mathsf{J}}} z$ such that $z \preceq x_1 \prec x$ (since $w' \preceq x$ and $x_1 \prec x$).

Finally, we can see by Expand that $x = (a \cdot x_1)^* \mathrel{\mathsf{PB}^*} 1 + a \cdot z = y$. Since each $1, a, z \preceq x$, we have $y = 1 + a \cdot z \preceq x$ as needed.

($w \prec a$) Since $w \prec a$, we can apply our IH on (2) with our subterm order decreasing on $w \prec x$ to find that $w^* \mathrel{\mathsf{PB}^*} w'$ such that $w' \preceq w \prec a \preceq x$. By our IH on (1) with our subterm order decreasing on $w' \prec x$ to find that $w' \cdot x_1 \mathrel{\mathsf{PB}^{\mathsf{J}}} z$ where $z \preceq x$ (because $w' \preceq x$ and $x_1 \prec x$).

We can now see by Slide that $x = (a \cdot x_1)^* \mathrel{\mathsf{PB}^*} 1 + a \cdot z = y$. Since each $1, a, z \preceq x$, we have $y = 1 + a \cdot z \preceq x$ as needed.

(x_2 isn't vacuous) We have $x \equiv a \cdot x_1 + x_2$ where $x_i \prec x$ and $a \in mt(x)$. Since x_2 isn't vacuous, we must have $a \prec x$, not just $a \preceq x$.

By the IH on (2) with the subterm ordering decreasing on $x_2 \prec x$, we find $x_2 \mathrel{\mathsf{PB}^*} w$ such that $w \preceq x_2$. By the IH on (1) with the subterm ordering decreasing on $x_1 \prec x$, we have $x_1 \cdot w \mathrel{\mathsf{PB}^{\mathsf{J}}} v$ where $v \preceq x$ (because $x_1 \preceq x$ and $w \preceq x$). By the IH on (2) with the subterm ordering decreasing on $a \cdot v \prec x$, we find $(a \cdot v)^* \mathrel{\mathsf{PB}^*} z$ where $z \preceq a \cdot v \prec x$. By our IH on (1) with the subterm ordering decreasing on $w \prec x$, we find $w \cdot z \mathrel{\mathsf{PB}^{\mathsf{J}}} y$ where $y \prec x$ (because $w \prec x$ and $z \prec x$).

By Denest, we can see that $x \equiv (a \cdot x_1 + x_2)^* \mathrel{\mathsf{PB}^*} y$, and we've already found that $y \preceq x$ as needed.

Pushing tests through actions ($m \cdot a \mathrel{\mathsf{PB}^{\bullet}} y$). We go by cases on a and m to find the $y \preceq a$ such that $m \cdot a \mathrel{\mathsf{PB}^{\bullet}} y$.

($m, 0$) We have $m \cdot 0 \mathrel{\mathsf{PB}^{\bullet}} 0$ by SeqZero, and $0 \preceq 0$ immediately.

($m, 1$) We have $m \cdot 1 \mathrel{\mathsf{PB}^{\bullet}} 1 \cdot m$ by SeqOne and $1 \preceq 1$ immediately.

($m, a \cdot b$) By the IH on (3) decreasing in size on a, we know that $m \cdot a \mathrel{\mathsf{PB}^{\bullet}} x$ where $x \preceq a \preceq a \cdot b$. By the IH on (5) decreasing in size on b, we know that $x \cdot b \mathrel{\mathsf{PB}^{\mathsf{T}}} y$. Finally, we know by SeqSeqTest that $m \cdot (a \cdot b) \mathrel{\mathsf{PB}^{\bullet}} y$. Since $x \preceq a \cdot b$ and $b \preceq a \cdot b$, we know by the IH on (5) earlier that $y \preceq a \cdot b$.

$(m, a + b)$ By the IH on (3) decreasing in size on a, we know that $m \cdot a$ PB$^\bullet$ x such that $x \preceq a \preceq a + b$. Similarly, by the IH on (3) decreasing in size on b, we know that $m \cdot b$ PB$^\bullet$ z such that $z \preceq b \preceq a + b$. By SeqParTest, we know that $m \cdot (a + b)$ PB$^\bullet$ $x + z = y$; by normal form parallel congruence, we know that $y = x + z \preceq a + b$ as needed.

$(m \cdot n, a)$ By the IH on (3) decreasing in size on n, we know that $n \cdot a$ PB$^\bullet$ x such that $x \preceq a$. By the IH on (4) decreasing in size on m, we know that $m \cdot x$ PB$^\mathsf{R}$ y such that $y \preceq x \preceq a$ (which are the size bounds on y we needed to show). All that remains to be seen is that $(m \cdot n) \cdot a$ PB$^\bullet$ y, which we have by SeqSeqAction.

$(m + n, a)$ By the IH on (3) decreasing in size on m, we know that $m \cdot a$ PB$^\bullet$ x. Similarly, by the IH on (3) decreasing in size on n, we know that $n \cdot a$ PB$^\bullet$ z. By SeqParAction, we know that $(m + n) \cdot a$ PB$^\bullet$ $x + z = y$. Furthermore, both IHs let us know that $x, z \preceq a$, so by normal form parallel congruence, we know that $y = x + z \preceq a$.

$(\pi, \neg a)$ By the IH on (3) decreasing in size on a, we can find that $\pi \cdot a$ PB$^\bullet$ $\sum_i a_i \cdot \pi$ where $\sum_i a_i \preceq a$, and $\mathsf{nnf}(\neg(\sum_i a_i)) = b$ for some term b. It remains to be seen that $b \preceq \neg a$, which we have by monotonicity of nnf (Lemma 3.21).

(π, α) In this case, we fall back on the client theory's pushback operation (Definition 3.26). We have $\pi \cdot \alpha$ WP $\{a_1, \ldots, a_k\}$ such that $a_i \preceq \alpha$. By Prim, we have $\pi \cdot \alpha$ PB$^\bullet$ $\sum_{i=1}^k a_i \cdot \pi = y$; since each $a_i \preceq \alpha$, we find $y \preceq \alpha$ by the monotonicity of union (Lemma 3.18).

(m^*, a) We've already ruled out the case where $a = b \cdot c$, so it must be the case that $\mathsf{seqs}(a) = \{a\}$, so $\mathsf{mt}(a) = \{a\}$.

By the IH on (3) decreasing in size on m, we know that $m \cdot a$ PB$^\bullet$ x such that $x \preceq a$. There are now two possibilities: either $x \prec a$ or $a \in \mathsf{mt}(x) = \{a\}$.

$(x \prec a)$ By the IH on (4) with $x \prec a$, we know by SeqStarSmaller that $m^* \cdot x$ PB$^\mathsf{R}$ y such that $y \preceq x \prec a$.

$(a \in \mathsf{mt}(x))$ By splitting (Lemma 3.25), we have $x \equiv a \cdot t + u$, where t and u are normal forms such that $t, u \prec x \preceq a$.

By the IH on (4) with $t \prec a$, we know that $m^* \cdot t$ PB$^\mathsf{R}$ w such that $w \preceq t \prec x \preceq a$. By the IH on (2) with $u \prec x \preceq a$, we know that u^* PB* z such that $z \preceq u \prec x \preceq a$. By the IH on (1) with $w \prec a$ and $z \prec a$, we find that $w \cdot z$ PB$^\mathsf{J}$ v such that $v \preceq w \prec a$.

Finally we have our y: by SeqStarInv, we have $m^* \cdot a$ PB$^\bullet$ $a \cdot z + v = y$. Since $z \preceq a$ and $a \preceq a$, we have $a \cdot z \preceq a$ (mixed sequence congruence; Lemma 3.22) and $v \prec a$. By normal form parallel congruence, we have $a \cdot z + v \preceq a$ (Lemma 3.22).

Pushing normal forms through actions ($m \cdot x$ PB$^\mathsf{R}$ z). We have $x = \sum_{i=1}^k a_i \cdot n_i$; by the IH on (3) with the size decreasing on n_i, we know that $m \cdot a_i$ PB$^\bullet$ x_i for each i such that $x_i \preceq a_i$, so by Restricted, we know that $m \cdot x$ PB$^\mathsf{R}$ $\sum_{i=1}^k x_i n_i = y$.

We must show that $y \preceq x$. By our IH on (3) we found earlier that $x_i \preceq a_i$. By normal form parallel congruence (Lemma 3.22), we have $y \preceq x$.

Pushing tests through normal forms ($x \cdot a$ PB$^\mathsf{T}$ y). We have $x = \sum_{i=1}^k a_i \cdot m_i$; by the IH on (3) with the size decreasing on m_i, we know that $m_i \cdot a$ PB$^\bullet$ $y_i = \sum_{j=1}^l b_{ij} m_{ij}$ where $y_i \preceq a$. Therefore, we know that $x \cdot a$ PB$^\mathsf{T}$ $\sum_{i=1}^k \sum_{j=1}^l a_i \cdot b_{ij} \cdot m_{ij} = y$ by Test.

Given that $x \leq z$ and $a \leq z$, We must show that $y \leq z$. We already know that $a_i \leq x \leq z$, and we found from the IH on (3) earlier that $b_{ij} \leq y_i \leq a \leq z$. By test bounding (Lemma 3.22), we have $a_i \cdot b_{ij} \leq z$, and therefore $y \leq z$ by normal form parallel congruence (Lemma 3.22). □

Finally, to reiterate our discussion of $\mathsf{PB}^{\bullet}$, Theorem 3.35 shows that every left-hand side of the pushback relation has a corresponding right-hand side. We *haven't* proved that the pushback relation is functional— if a term has more than one maximal test, there could be many different choices of how we perform the pushback.

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

Corollary 3.36 (Normal forms). *For all $p \in \mathcal{T}^*$, there exists a normal form x such p norm x and that $p \equiv x$.*

Proof. By induction on p.

- (Pred) We have $a \equiv a$ immediately.
- (Act) We have $\pi \equiv 1 \cdot \pi$ by KA-Seq-One.
- (Par) By the IHs and congruence.
- (Seq) We have $p = q \cdot r$; by the IHs, we know that q norm x and r norm y. By pushback existence (Theorem 3.35), we know that $x \cdot y \ \mathsf{PB}^{\mathsf{J}} \ z$ for some z. By pushback soundness (Theorem 3.34), we know that $x \cdot y \equiv z$. By congruence, $p \equiv z$.
- (Star) We have $p = q^*$. By the IH, we know that q norm x. By pushback existence (Theorem 3.35), we know that $x^* \ \mathsf{PB}^* \ y$. By pushback soundness (Theorem 3.34), we know that $x^* \equiv y$.

□

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 3.34 and 3.35 as a single theorem and without any explicit normalization or pushback relation.

3.4 Completeness

We prove completeness—if $[\![p]\!] = [\![q]\!]$ then $p \equiv q$—by normalizing p and q and comparing the resulting terms. Our completeness proof uses the completeness of Kleene algebra (KA) as its foundation: the set of possible traces of actions performed for a restricted (test-free) action in our denotational semantics is a regular language, and so the KA axioms are sound and complete for it. In order to relate our denotational semantics to regular languages, we define the regular interpretation of restricted actions $m \in \mathcal{T}_{\mathrm{RA}}$ in the conventional way and then relate our denotational semantics to the regular interpretation (Fig. 17). 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, 30]. We haven't attempted the proof or an implementation.

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

Fig. 17. Regular interpretation of restricted actions

Lemma 3.37 (Restricted actions are ahistorical). *If* $[\![m]\!](t_1) = t_1, t$ *and* $\mathsf{last}(t_1) = \mathsf{last}(t_2)$ *then* $[\![m]\!](t_2) = t_2, t$.

Proof. By induction on m.

$(m = 1)$ Immediate, since t is empty.

$(m = \pi)$ We immediately have $t = \langle \mathsf{last}(t_1), \pi \rangle$.

$(m = m + n)$ We have $[\![m+n]\!](t_1) = [\![m]\!](t_1) \cup [\![n]\!](t_1)$ and $[\![m+n]\!](t_2) = [\![m]\!](t_2) \cup [\![n]\!](t_2)$. By the IHs.

$(m = m \cdot n)$ We have $[\![m \cdot n]\!](t_1) = ([\![m]\!] \bullet [\![n]\!])(t_1)$ and $[\![m \cdot n]\!](t_2) = ([\![m]\!] \bullet [\![n]\!])(t_2)$. It must be that $[\![m]\!](t_1) = \{t_1, t_{mi}\}$, so by the IH we have $[\![m]\!](t_2) = \{t_2, t_{mi}\}$. These sets have the same last states, so we can apply the IH again for n, and we are done.

$(m = m^*)$ We have $[\![m^*]\!](t_1) = \bigcup_{0 \leq i} [\![m]\!]^i(t_1)$. By induction on i.

$(i = 0)$ Immediate, since $[\![m]\!]^0(t_i) = t_i$ and so t is empty.

$(i = i + 1)$ By the IH and the reasoning above for $\cdot$.

□

Lemma 3.38 (Labels are regular). $\{\mathsf{label}([\![m]\!](\langle \sigma, \bot \rangle)) \mid \sigma \in \mathsf{State}\} = \mathcal{R}(m)$

Proof. By induction on the restricted action m.

$(m = 1)$ We have $\mathcal{R}(1) = \{\epsilon\}$. For all σ, we find $[\![1]\!](\langle \sigma, \bot \rangle) = \{\langle \sigma, \bot \rangle\}$, and $\mathsf{label}(\langle \sigma, \bot \rangle) = \epsilon$.

$(m = \pi)$ We $\mathcal{R}(\pi) = \{\pi\}$. For all σ, we find $[\![\pi]\!](\langle \sigma, \bot \rangle) = \{\langle \sigma, \bot \rangle \langle \mathsf{act}(\pi, \sigma), \pi \rangle\}$, and so $\mathsf{label}(\langle \sigma, \bot \rangle \langle \mathsf{act}(\pi, \sigma), \pi \rangle) = \pi$.

$(m = m + n)$ We have $\mathcal{R}(m + n) = \mathcal{R}(m) \cup \mathcal{R}(n)$. For all σ, we have:

$$
\begin{array}{rcl}
\mathsf{label}([\![m+n]\!](\langle \sigma, \bot \rangle)) & = & \mathsf{label}([\![m]\!](\langle \sigma, \bot \rangle) \cup [\![n]\!](\langle \sigma, \bot \rangle)) \\
 & = & \mathsf{label}([\![m]\!](\langle \sigma, \bot \rangle)) \cup \mathsf{label}([\![n]\!](\langle \sigma, \bot \rangle))
\end{array}
$$

and we are done by the IHs.

$(m = m \cdot n)$ We have $\mathcal{R}(m \cdot n) = \{uv \mid u \in \mathcal{R}(m), v \in \mathcal{R}(n)\}$. For all σ, we have:

$$
\begin{array}{rcll}
\mathsf{label}([\![m \cdot n]\!](\langle \sigma, \bot \rangle)) & = & \mathsf{label}(([\![m]\!] \bullet [\![n]\!])(\langle \sigma, \bot \rangle)) & \\
 & = & \mathsf{label}(\bigcup_{t \in [\![m]\!](\langle \sigma, \bot \rangle)} \mathsf{label}([\![n]\!](t))) & \\
 & = & \mathsf{label}(\bigcup_{t \in [\![m]\!](\langle \sigma, \bot \rangle)} \mathsf{label}(t[\![n]\!](\langle \sigma, \bot \rangle))) & \text{by Lemma 3.37} \\
 & = & \mathsf{label}([\![m]\!](\langle \sigma, \bot \rangle))\mathsf{label}([\![n]\!](\langle \sigma, \bot \rangle)) &
\end{array}
$$

and we are done by the IHs.

$(m = m^*)$ We have $\mathcal{R}(m^*) = \bigcup_{0 \leq i} \mathcal{R}(m)^i$. For all σ, we have:

$$
\begin{array}{rcl}
\mathsf{label}([\![m^*]\!](\langle \sigma, \bot \rangle)) & = & \mathsf{label}(\bigcup_{0 \leq i} [\![m]\!]^i(\langle \sigma, \bot \rangle)) \\
 & = & \bigcup_{0 \leq i} \mathsf{label}([\![m]\!]^i(\langle \sigma, \bot \rangle))
\end{array}
$$

and we are done by the IH.

□

Our proof of completeness works by normalizing each side of the equation, making each side locally unambiguous, making the entire equation unambiguous, and then using word equality to ensure that normal forms with equivalent predicates have equivalent actions.

Theorem 3.39 (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 3.36); by soundness (Theorem 3.5), 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'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 (excluding the case where we select none) and run the actions corresponding to the true predicates, i.e., m_i gets run precisely when a_i is true:

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

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 o_j could safely be different. We have assumed that the predicates of $\mathcal{T}$ can be checked for emptiness, so 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 each label of these commands must be equal—and so $\mathcal{R}(l_i) = \mathcal{R}(o_j)$ (Lemma 3.38). By the deductive completeness of KA, we know that KA $\vdash l_i \equiv o_j$. Since we have the KA axioms in our system, then $l_i \equiv o_j$; by reflexivity, we know that $c_i \cdot d_j \equiv c_i \cdot d_j$, and we have proved that $\ddot{x} \equiv \ddot{y}$. By transitivity, we can see that $\hat{x} \equiv \hat{y}$ and so $x \equiv y$ and $p \equiv q$, as desired. □

4 IMPLEMENTATION

We have implemented our ideas in an OCaml library; the library's source code, tests, and our evaluation workbench are available online.[1] Sec. 1.3 summarizes the high-level idea and gives an example library implementation for the theory of increasing natural numbers. To use a higher-order theory such as that of product theories, one need only instantiate the appropriate modules in the library:

```
module P = Product(IncNat)(Boolean)
module 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 (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. 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 equivalence function. Our implementation currently supports both a decision procedure based on automata (not yet completely correct, and so omitted from this article) and one based on the normalization term-rewriting from the completeness proof. We've implemented a command-line tool that can be configured to work these theories; given a variety of KMT terms as input, it partitions them into equivalence classes using the decision procedure of the user's choice.

4.1 Optimizations

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.

Our hash-consing constructors are *smart* constructors, automatically rewriting common identities (e.g., constructing $p \cdot 1$ will simply return p; constructing $(p^*)^*$ will simply return p^*). Client theories can extend our smart constructors to witness theory-specific identities. These optimizations are partly responsible for the speed of our normalization routine (when it avoids the costly Denest case). When deciding equivalence using normalization, we use the Hopcroft and Karp algorithm [32] on implicit automata using the Brzozowski derivative [9] to generate the transition relation on-the-fly.

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

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
$x = \mathsf{f}; (\mathsf{flip}\ x; \mathsf{flip}\ x)^* \equiv (\mathsf{flip}\ x; \mathsf{flip}\ x)^*; x = \mathsf{f}$	$\mathcal{B}$	<0.001s
$w := \mathsf{f}; x := \mathsf{t}; y := \mathsf{f}; z := \mathsf{f};$ $((w = \mathsf{t} + x = \mathsf{t} + y = \mathsf{t} + z = \mathsf{t}); a := \mathsf{t}+$ $(\neg(w = \mathsf{t} + x = \mathsf{t} + y = \mathsf{t} + z = \mathsf{t})); a := \mathsf{f})$ $\equiv\ w := \mathsf{f}; x := \mathsf{t}; y := \mathsf{f}; z := \mathsf{f};$ $(((w = \mathsf{t} + x = \mathsf{t}) + (y = \mathsf{t} + z = \mathsf{t})); a := \mathsf{t}+$ $(\neg((w = \mathsf{t} + x = \mathsf{t}) + (y = \mathsf{t} + z = \mathsf{t}))); a := \mathsf{f})$	$\mathcal{B}$	<0.001s
$y < 1; a = \mathsf{t}; \mathsf{inc}_y;$ $(1 + b = \mathsf{t}; \mathsf{inc}_y);$ $(1 + c = \mathsf{t}; \mathsf{inc}_y); y > 2$ $\equiv\ y < 1; a = \mathsf{t}; b = \mathsf{t}; c = \mathsf{t}; \mathsf{inc}_y; \mathsf{inc}_y; \mathsf{inc}_y$	$\mathbb{N} \times \mathcal{B}$	0.309s
$(\mathsf{flip}\ x + \mathsf{flip}\ y + \mathsf{flip}\ z)* = (\mathsf{flip}\ x + \mathsf{flip}\ y + \mathsf{flip}\ z)*$	$\mathbb{B}$	>30s (timeout)

Fig. 18. Implementation microbenchmarks

Client theories can implement custom solvers or rely on Z3 embeddings—custom solvers are typically faster. We've implemented a few of these domain-specific optimizations: satisfiability procedure for `IncNat` makes a heuristic decision between using our incomplete custom solver or Z3 [18]—our solver is much faster on its restricted domain.

5 EVALUATION

We performed a few experiments to evaluate our tool on a collection of simple microbenchmarks. Fig. 18 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 . We use a timeout of thirty seconds.

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, 37]) 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 fourth, parity-swapping benchmark is one such example – it flips the parity of a boolean variable an even number of times 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.

6 RELATED WORK

Kozen and Mamouras's Kleene algebra with equations [40] 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 trace 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, 30]; 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. 2). It may be possible to build a hybrid framework, with ideas from both. A trace semantics occurs in previous work on KAT as well [26, 37].

Kozen studies KATs with arbitrary equations $x := e$ [38], also called Schematic KAT, where e comes from arbitrary first-order structures over a fixed signature Σ. He has a pushback-like axiom $x := e \cdot p \equiv \phi[{}^{x}/_{e}] \cdot x := e$. Arbitrary first-order structures over Σ's theory are much more expressive than anything we can handle—the pushback may or may not decrease in size, depending on Σ; KATs over such theories are generally undecidable. We, on the other hand, are able to offer pay-as-you-go results for soundness and completeness as well as an 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. [42] 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 [39, 54, 59], 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.

Smolka et al. [61] 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, while our KMTs are completely concrete.

Our work is loosely related to Satisfiability Modulo Theories (SMT) [19]. The high-level motivation is the same—to create an extensible framework where custom theories can be combined [47] and used to increase the expressiveness and power [62] of the underlying technique (SAT vs. KA). However, the specifics vary greatly—while SMT is used to reason about the formula satisfiability, KMT is used to reason about how program structure interacts with tests. 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, 55]. 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, 34, 49, 57], 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.

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 trace semantics; we derive 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 [30], natural numbers, unbounded setsand maps, networks [1], and temporal logic [8]. Our ability to reason about unbounded state is novel. Our work, however, is limited to tracing semantics; we conjecture that it is possible to merge actions (as in KAT+B!, NetKAT, and Kleene algebra with equations [1, 30, 40]), but leave it to future work.

ACKNOWLEDGMENTS

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

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] M Anderson. 2014. Time Warner Cable Says Outages Largely Resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved.

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

[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] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481âĂŞ494. https://doi.org/10.1145/321239.321249

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

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

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

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

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

[15] 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* (Saarbrücken, 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] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *SIGCOMM*.
[29] Joanne Godfrey. 2016. The Summer of Network Misconfigurations. https://goo.gl/ViU9uS.
[30] 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.
[31] 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
[32] John E. Hopcroft and R. M. Karp. 1971. *A Linear Algorithm for Testing Equivalence of Finite Automata*. Technical Report 71-114. Cornell University.
[33] Zeus Kerravala. 2004. What is Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. https://www.cs.princeton.edu/courses/archive/fall10/cos561/papers/Yankee04.pdf.
[34] 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.
[35] Dexter Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Inf. Comput.* 110, 2 (1994), 366–390. https://doi.org/10.1006/inco.1994.1037
[36] 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
[37] Dexter Kozen. 2003. *Kleene algebra with tests and the static analysis of programs*. Technical Report. Cornell University.
[38] 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).
[39] Dexter Kozen. 2017. On the Coalgebraic Theory of Kleene Algebra with Tests. In *Rohit Parikh on Logic, Language and Society*. Springer, 279–298.

[40] 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.
[41] 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.
[42] Kim G Larsen, Stefan Schmid, and Bingtian Xue. 2016. WNetKAT: Programming and Verifying Weighted Software-Defined Networks. In *OPODIS*.
[43] Ratul Mahajan, David Wetherall, and Tom Anderson. 2002. Understanding BGP Misconfiguration. In *SIGCOMM*.
[44] 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.
[45] 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
[46] Yoshiki Nakamura. 2015. Decision Methods for Concurrent Kleene Algebra with Tests: Based on Derivative. *RAMiCS 2015* (2015), 1.
[47] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245–257.
[48] Juniper Networks. 2008. As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management. https://www-935.ibm.com/services/au/gts/pdf/200249.pdf.
[49] 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.
[50] 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.
[51] 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
[52] Yakov Rekhter and Tony Li. 1995. *A Border Gateway Protocol 4 (BGP-4)*. RFC 1654. RFC Editor. 1–56 pages. http://www.rfc-editor.org/rfc/rfc1654.txt
[53] Grigore Roşu. 2016. Finite-Trace Linear Temporal Logic: Coinductive Completeness. In *International Conference on Runtime Verification*. Springer, 333–350.
[54] 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.
[55] Andrew E. Santosa. 2015. Comparing Weakest Precondition and Weakest Liberal Precondition. *CoRR* abs/1512.04013 (2015).
[56] 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.
[57] 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.
[58] Simon Sharwood. 2016. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/.
[59] Alexandra Silva. 2010. *Kleene Coalgebra*. PhD Thesis. University of Minho, Braga, Portugal.
[60] 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.
[61] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2019. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time. *Proc. ACM Program. Lang.* 4, POPL, Article 61 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371129

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

[63] Yevgenly Sverdlik. 2012. Microsoft: misconfigured network device led to Azure outage. https://goo.gl/Se7DzD