Guarded Kleene Algebra with Tests

Verification of Uninterpreted Programs in Nearly Linear Time

STEFFEN SMOLKA, Cornell University, USA
NATE FOSTER, Cornell University, USA
JUSTIN HSU, University of Wisconsin–Madison, USA
TOBIAS KAPPÉ, University College London, UK
DEXTER KOZEN, Cornell University, USA
ALEXANDRA SILVA, University College London, UK

Guarded Kleene Algebra with Tests (GKAT) is a variation on Kleene Algebra with Tests (KAT) that arises by restricting the union (+) and iteration (∗) operations from KAT to predicate-guarded versions. We develop the (co)algebraic theory of GKAT and show how it can be efficiently used to reason about imperative programs. In contrast to KAT, whose equational theory is PSPACE-complete, we show that the equational theory of GKAT is (almost) linear time. We also provide a full Kleene theorem and prove completeness for an analogue of Salomaa's axiomatization of Kleene Algebra.

CCS Concepts: • **Theory of computation → Program schemes**; **Program reasoning**.

Additional Key Words and Phrases: uninterpreted programs, program equivalence, program schemes, guarded automata, coalgebra, Kleene algebra with Tests

ACM Reference Format:
Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2020. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time. *Proc. ACM Program. Lang.* 4, POPL, Article 61 (January 2020), 28 pages. https://doi.org/10.1145/3371129

> This paper is dedicated to Laurie J. Hendren (1958–2019), whose passion for teaching and research inspired us and many others. Laurie's early work on McCAT [Erosa and Hendren 1994] helped us understand the limits of Guarded Kleene Algebra with Tests and devise a suitable definition of *well-nestedness* that underpins our main results.

1 INTRODUCTION

Computer scientists have long explored the connections between families of programming languages and abstract machines. This dual perspective has furnished deep theoretical insights as well as practical tools. As an example, Kleene's classic result establishing the equivalence of regular expressions and finite automata [Kleene 1956] inspired decades of work across a variety of areas including programming language design, mathematical semantics, and formal verification.

Authors' addresses: Steffen Smolka, Cornell University, Ithaca, NY, USA, smolka@cs.cornell.edu; Nate Foster, Cornell University, Ithaca, NY, USA, jnfoster@cs.cornell.edu; Justin Hsu, University of Wisconsin–Madison, Madison, WI, USA, email@justinh.su; Tobias Kappé, University College London, London, UK, tkappe@cs.ucl.ac.uk; Dexter Kozen, Cornell University, Ithaca, NY, USA, kozen@cs.cornell.edu; Alexandra Silva, University College London, London, UK, alexandra.silva@ucl.ac.uk.

2475-1421/2020/1-ART61
https://doi.org/10.1145/3371129

Kleene Algebra with Tests (KAT) [Kozen 1996], which combines Kleene Algebra (KA) with Boolean Algebra (BA), is a modern example of this approach. Viewed from the program-centric perspective, a KAT models the fundamental constructs that arise in programs: sequencing, branching, iteration, etc. The equational theory of KAT enables algebraic reasoning and can be finitely axiomatized [Kozen and Smith 1996]. Viewed from the machine-centric perspective, a KAT describes a kind of automaton that generates a regular language of traces. This shift in perspective admits techniques from coalgebra for reasoning about program behavior. In particular, there are efficient algorithms for checking bisimulation, which can be optimized using properties of bisimulations [Bonchi and Pous 2013; Hopcroft and Karp 1971] or symbolic automata representations [Pous 2015].

KAT has been used to model computation across a wide variety of areas including program transformations [Angus and Kozen 2001; Kozen 1997], concurrency control [Cohen 1994b], compiler optimizations [Kozen and Patron 2000], cache control [Barth and Kozen 2002; Cohen 1994a], and more [Cohen 1994a]. A prominent recent application is NetKAT [Anderson et al. 2014], a language for reasoning about the packet-forwarding behavior of software-defined networks. NetKAT has a sound and complete equational theory, and a coalgebraic decision procedure that can be used to automatically verify many important networking properties including reachability, loop-freedom, and isolation [Foster et al. 2015]. However, while NetKAT's implementation scales well in practice, deciding equivalence for NetKAT is PSPACE-complete in the worst case [Anderson et al. 2014].

A natural question to ask is whether there is an efficient fragment of KAT that is reasonably expressive, while retaining a solid foundation. We answer this question positively with a comprehensive study of Guarded Kleene Algebra with Tests (GKAT), the guarded fragment of KAT. GKAT is a propositional abstraction of imperative while programs. We establish the fundamental properties of GKAT and develop its algebraic and coalgebraic theory. GKAT replaces the union ($e + f$) and iteration (e^*) constructs in KAT with guarded versions: conditionals ($e +_b f$) and loops ($e^{(b)}$) guarded by Boolean predicates b. The resulting language is a restriction of full KAT, but sufficiently expressive to model typical, imperative programs—*e.g.*, essentially all NetKAT programs needed to solve practical verification problems can be expressed as guarded programs.

In exchange for a modest sacrifice in expressiveness, GKAT offers two significant advantages. First, program equivalence (for a fixed Boolean algebra) is decidable in *nearly linear time*—a substantial improvement over the PSPACE complexity for KAT [Cohen et al. 1996]. Specifically, any GKAT expression e can be represented as a deterministic automaton of size $O(|e|)$, while KAT expressions can require as many as $O(2^{|e|})$ states. As a consequence, any language property that is efficiently decidable for deterministic automata is also efficiently decidable for GKAT. Second, we believe that GKAT is a better foundation for probabilistic languages due to well-known issues that arise when combining non-determinism—which is native to KAT—with probabilistic choice [Mislove 2006; Varacca and Winskel 2006]. For example, ProbNetKAT [Foster et al. 2016], a probabilistic extension of NetKAT, does not satisfy the KAT axioms, but its guarded restriction forms a proper GKAT.

Although GKAT is a simple restriction of KAT at the syntactic level, its semantics is surprisingly subtle. In particular, the "obvious" notion of GKAT automata can encode behaviors that would require non-local control-flow operators (e.g, **goto** or multi-level **break** statements) [Kozen and Tseng 2008]. In contrast, GKAT models programs whose control-flow always follows a lexical, nested structure. To overcome this discrepancy, we identify restrictions on automata to enable an analogue of Kleene's theorem—every GKAT automaton satisfying our restrictions can be converted to a program, and vice versa. Besides the theoretical interest in this result, we believe it may also have practical applications, such as reasoning about optimizations in a compiler [Hendren et al. 1992]. We also develop an equational axiomatization for GKAT and prove that it is sound and complete over a coequationally-defined language model. The main challenge is that without +, the

natural order on KAT programs can no longer be used to axiomatize a least fixpoint. We instead axiomatize a unique fixed point, in the style of Salomaa's work on Kleene Algebra [Salomaa 1966].

Outline. We make the following contributions in this paper.

- We initiate a comprehensive study of GKAT, a guarded version of KAT, and show how GKAT models relational and probabilistic programming languages (§ 2).
- We give a new construction of linear-size automata from GKAT programs (§ 4). As a consequence, the equational theory of GKAT (over a fixed Boolean algebra) is decidable in nearly linear time (§ 5).
- We identify a class of automata representable as GKAT expressions (§ 4) that contains all automata produced by the previous construction, yielding a Kleene theorem.
- We present axioms for GKAT (§ 3) and prove that our axiomatization is complete for equivalence with respect to a coequationally-defined language model (§ 6).

Omitted proofs appear in the appendix of the extended version of this paper [Smolka et al. 2019a].

2 OVERVIEW: AN ABSTRACT PROGRAMMING LANGUAGE

This section introduces the syntax and semantics of GKAT, an abstract programming language with uninterpreted actions. Using examples, we show how GKAT can model relational and probabilistic programming languages—*i.e.*, by giving actions a concrete interpretation. An equivalence between abstract GKAT programs thus implies a corresponding equivalence between concrete programs.

2.1 Syntax

The syntax of GKAT is parameterized by abstract sets of *actions* Σ and *primitive tests* T, where Σ and T are assumed to be disjoint and nonempty, and T is assumed to be finite. We reserve p and q to range over actions, and t to range over primitive tests. The language consists of Boolean expressions, BExp, and GKAT expressions, Exp, as defined by the following grammar:

$b, c, d \in \mathsf{BExp} ::=$		$e, f, g \in \mathsf{Exp} ::=$	
$\mid 0$	**false**	$\mid p \in \Sigma$	**do** p
$\mid 1$	**true**	$\mid b \in \mathsf{BExp}$	**assert** b
$\mid t \in T$	t	$\mid e \cdot f$	$e; f$
$\mid b \cdot c$	b **and** c	$\mid e +_b f$	**if** b **then** e **else** f
$\mid b + c$	b **or** c	$\mid e^{(b)}$	**while** b **do** e
$\mid \overline{b}$	**not** b		

The algebraic notation on the left is more convenient when manipulating terms, while the notation on the right may be more intuitive when writing programs. We often abbreviate $e \cdot f$ by ef, and omit parentheses following standard conventions, *e.g.*, writing $bc + d$ instead of $(bc) + d$ and $ef^{(b)}$ instead of $e(f^{(b)})$.

2.2 Semantics: Language Model

Intuitively, we interpret a GKAT expression as the set of "legal" execution traces it induces, where a trace is legal if no assertion fails. To make this formal, let $b \equiv_{\mathrm{BA}} c$ denote Boolean equivalence. Entailment is a preorder on the set of Boolean expressions, BExp, and can be characterized in terms of equivalence as follows: $b \leq c \iff b + c \equiv_{\mathrm{BA}} c$. In the quotient set $\mathsf{BExp}/{\equiv_{\mathrm{BA}}}$ (the *free Boolean algebra* on generators $T = \{t_1, \ldots, t_n\}$), entailment is a partial order $[b]_{\equiv_{\mathrm{BA}}} \leq [c]_{\equiv_{\mathrm{BA}}} :\iff b+c \equiv_{\mathrm{BA}} c$, with minimum and maximum elements given by the equivalence classes of 0 and 1, respectively. The minimal nonzero elements of this order are called *atoms*. We let At denote the set of atoms and use lowercase Greek letters $\alpha, \beta, \ldots$ to denote individual atoms. Each atom is the equivalence class

of an expression of the form $c_1 \cdot c_2 \cdots c_n \in \mathsf{BExp}$ with $c_i \in \{t_i, \overline{t_i}\}$. Thus we can think of each atom as representing a truth assignment on T, *e.g.*, if $c_i = t_i$ then t_i is set to true, otherwise if $c_i = \overline{t_i}$ then t_i is set to false. Likewise, the set $\{\alpha \in \mathsf{At} \mid \alpha \leq b\}$ can be thought of as the set of truth assignments where b evaluates to true; $\equiv_{\mathrm{BA}}$ is *complete* with respect to this interpretation in that two Boolean expressions are related by $\equiv_{\mathrm{BA}}$ if and only if their atoms coincide [Birkhoff and Bartee 1970].

A *guarded string* is an element of the regular set $\mathsf{GS} \coloneqq \mathsf{At} \cdot (\Sigma \cdot \mathsf{At})^*$. Intuitively, a non-empty string $\alpha_0 p_1 \alpha_1 \cdots p_n \alpha_n \in \mathsf{GS}$ describes a trace of an abstract program: the atoms α_i describe the state of the system at various points in time, starting from an initial state α_0 and ending in a final state α_n, while the actions $p_i \in \Sigma$ are the transitions triggered between the various states. Given two traces, we can combine them sequentially by running one after the other. Formally, guarded strings compose via a partial *fusion product* $\diamond\colon \mathsf{GS} \times \mathsf{GS} \rightharpoonup \mathsf{GS}$, defined for $x, y \in (\mathsf{At} \cup \Sigma)^*$ as

$$x\alpha \diamond \beta y \coloneqq \begin{cases} x\alpha y & \text{if } \alpha = \beta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This product lifts to a total function on languages $L, K \subseteq \mathsf{GS}$ of guarded strings, given by

$$L \diamond K \coloneqq \{x \diamond y \mid x \in L, y \in K\}.$$

We need a few more constructions before we can interpret GKAT expressions as languages representing their possible traces. First, 2^{GS} with the fusion product forms a monoid with identity At and so we can define the n-th power L^n of a language L inductively in the usual way:

$$L^0 \coloneqq \mathsf{At} \qquad\qquad L^{n+1} \coloneqq L^n \diamond L$$

Second, in the special case where $B \subseteq \mathsf{At}$, we write $\overline{B}$ for $\mathsf{At} - B$ and define:

$$L +_B K \coloneqq (B \diamond L) \cup (\overline{B} \diamond K) \qquad\qquad L^{(B)} \coloneqq \bigcup_{n \geq 0} (B \diamond L)^n \diamond \overline{B}$$

We are now ready to interpret GKAT expressions as languages of guarded strings via the semantic map $[\![-]\!] : \mathsf{Exp} \to 2^{\mathsf{GS}}$ as follows:

$$[\![p]\!] \coloneqq \{\alpha p \beta \mid \alpha, \beta \in \mathsf{At}\} \qquad [\![e \cdot f]\!] \coloneqq [\![e]\!] \diamond [\![f]\!] \qquad [\![e^{(b)}]\!] \coloneqq [\![e]\!]^{([\![b]\!])}$$
$$[\![b]\!] \coloneqq \{\alpha \in \mathsf{At} \mid \alpha \leq b\} \qquad [\![e +_b f]\!] \coloneqq [\![e]\!] +_{[\![b]\!]} [\![f]\!]$$

We call this the *language model* of GKAT. Since we make no assumptions about the semantics of actions, we interpret them as sets of traces beginning and ending in arbitrary states; this soundly overapproximates the behavior of any instantiation. A test is interpreted as the set of states satisfying the test. The traces of $e \cdot f$ are obtained by composing traces from e with traces from f in all possible ways that make the final state of an e-trace match up with the initial state of an f-trace. The traces of $e +_b f$ collect traces of e and f, restricting to e-traces whose initial state satisfies b and f-traces whose initial state satisfies $\overline{b}$. The traces of $e^{(b)}$ are obtained by sequentially composing zero or more be-traces and selecting traces ending in a state satisfying $\overline{b}$.

Remark 2.1 (Connection to KAT). The expressions for KAT, denoted KExp, are generated by the same grammar as for GKAT, except that KAT's union ($+$) replaces GKAT's guarded union ($+_b$) and KAT's iteration (e^*) replaces GKAT's guarded iteration ($e^{(b)}$). GKAT's guarded operators can be encoded in KAT; this encoding, which goes back to early work on Propositional Dynamic Logic [Fischer and Ladner 1979], is the standard method to model conditionals and while loops:

$$e +_b f \mapsto be + \overline{b}f \qquad\qquad e^{(b)} \mapsto (be)^* \overline{b}$$

Thus, there is a homomorphism $\varphi\colon \mathsf{Exp} \to \mathsf{KExp}$ from GKAT to KAT expressions. We inherit KAT's language model [Kozen and Smith 1996], $\mathcal{K}[\![-]\!]\colon \mathsf{KExp} \to 2^{\mathsf{GS}}$, in the sense that $[\![-]\!] = \mathcal{K}[\![-]\!] \circ \varphi$.

The languages denoted by GKAT programs satisfy an important property:

Definition 2.2 (Determinacy property). A language of guarded strings $L \subseteq \mathrm{GS}$ satisfies the *determinacy property* if, whenever string $x, y \in L$ agree on their first n atoms, then they agree on their first n actions (or lack thereof). For example, $\{\alpha p\gamma, \alpha p\delta, \beta q\delta\}$ and $\{\alpha p\gamma, \beta\}$ for $\alpha \neq \beta$ satisfy the determinacy property, while $\{\alpha p\beta, \alpha\}$ and $\{\alpha p\beta, \alpha q\delta\}$ for $p \neq q$ do not.

We say that two expressions e and f are *equivalent* if they have the same semantics—*i.e.*, if $[\![e]\!] = [\![f]\!]$. In the following sections, we show that this notion of equivalence

- is sound and complete for relational and probabilistic interpretations (§ 2.3 and 2.4),
- can be finitely and equationally axiomatized in a sound (§ 3) and complete (§ 6) way, and
- is efficiently decidable in time nearly linear in the sizes of the expressions (§ 4 and 5).

2.3 Relational Model

This subsection gives an interpretation of GKAT expressions as binary relations, a common model of input-output behavior for many programming languages. We show that the language model is sound and complete for this interpretation. Thus GKAT equivalence implies program equivalence for any programming language with a suitable relational semantics.

Definition 2.3 (Relational Interpretation). Let $i = (\mathrm{State}, \mathrm{eval}, \mathrm{sat})$ be a triple consisting of

- a set of *states* State,
- for each action $p \in \Sigma$, a binary relation $\mathrm{eval}(p) \subseteq \mathrm{State} \times \mathrm{State}$, and
- for each primitive test $t \in T$, a set of states $\mathrm{sat}(t) \subseteq \mathrm{State}$.

Then the *relational interpretation* of an expression e with respect to i is the smallest binary relation $\mathcal{R}_i[\![e]\!] \subseteq \mathrm{State} \times \mathrm{State}$ satisfying the following rules,

$$\frac{(\sigma,\sigma') \in \mathrm{eval}(p)}{(\sigma,\sigma') \in \mathcal{R}_i[\![p]\!]} \qquad \frac{\sigma \in \mathrm{sat}^\dagger(b)}{(\sigma,\sigma) \in \mathcal{R}_i[\![b]\!]} \qquad \frac{(\sigma,\sigma') \in \mathcal{R}_i[\![e]\!] \qquad (\sigma',\sigma'') \in \mathcal{R}_i[\![f]\!]}{(\sigma,\sigma'') \in \mathcal{R}_i[\![e \cdot f]\!]}$$

$$\frac{\sigma \in \mathrm{sat}^\dagger(b) \qquad (\sigma,\sigma') \in \mathcal{R}_i[\![e]\!]}{(\sigma,\sigma') \in \mathcal{R}_i[\![e +_b f]\!]} \qquad \frac{\sigma \in \mathrm{sat}^\dagger(\overline{b}) \qquad (\sigma,\sigma') \in \mathcal{R}_i[\![f]\!]}{(\sigma,\sigma') \in \mathcal{R}_i[\![e +_b f]\!]}$$

$$\frac{\sigma \in \mathrm{sat}^\dagger(b) \qquad (\sigma,\sigma') \in \mathcal{R}_i[\![e]\!] \qquad (\sigma',\sigma'') \in \mathcal{R}_i[\![e^{(b)}]\!]}{(\sigma,\sigma'') \in \mathcal{R}_i[\![e^{(b)}]\!]} \qquad \frac{\sigma \in \mathrm{sat}^\dagger(\overline{b})}{(\sigma,\sigma) \in \mathcal{R}_i[\![e^{(b)}]\!]}$$

where $\mathrm{sat}^\dagger : \mathrm{BExp} \to 2^{\mathrm{State}}$ is the usual lifting of $\mathrm{sat} : T \to 2^{\mathrm{State}}$ to Boolean expression over T.

The rules defining $\mathcal{R}_i[\![e]\!]$ are reminiscent of the big-step semantics of imperative languages, which arise as instances of the model for various choices of i. The following result says that the language model from the previous section abstracts the various relational interpretations in a sound and complete way. It was first proved for KAT by Kozen and Smith [1996].

Theorem 2.4. *The language model is sound and complete for the relational model:*

$$[\![e]\!] = [\![f]\!] \quad \Longleftrightarrow \quad \forall i.\, \mathcal{R}_i[\![e]\!] = \mathcal{R}_i[\![f]\!]$$

It is worth noting that Theorem 2.4 also holds for refinement (*i.e.*, with $\subseteq$ instead of $=$).

Example 2.5 (IMP). Consider a simple imperative programming language IMP with variable assignments and arithmetic and boolean expressions:

$$\begin{aligned} \textit{arithmetic expressions} \quad & a \in \mathcal{A} ::= x \in \mathsf{Var} \mid n \in \mathbb{Z} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\ \textit{boolean expressions} \quad & b \in \mathcal{B} ::= \mathbf{false} \mid \mathbf{true} \mid a_1 < a_2 \mid \mathbf{not}\ b \mid b_1\ \mathbf{and}\ b_2 \mid b_1\ \mathbf{or}\ b_2 \\ \textit{commands} \quad & c \in C ::= \mathbf{skip} \mid x := a \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \end{aligned}$$

IMP can be modeled in GKAT using actions for assignments and primitive tests for comparisons,[1]

$$\Sigma = \{x := a \mid x \in \mathsf{Var}, a \in \mathcal{A}\} \qquad T = \{a_1 < a_2 \mid a_1, a_2 \in \mathcal{A}\}$$

and interpreting GKAT expressions over the state space of variable assignments $\mathsf{State} := \mathsf{Var} \to \mathbb{Z}$:

$$\begin{aligned} \mathrm{eval}(x := a) &:= \{(\sigma, \sigma[x := n]) \mid \sigma \in \mathsf{State}, n = \mathcal{A}[\![a]\!](\sigma)\} \\ \sigma[x := n] &:= \lambda y. \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{else} \end{cases} \\ \mathrm{sat}(a_1 < a_2) &:= \{\sigma \in \mathsf{State} \mid \mathcal{A}[\![a_1]\!](\sigma) < \mathcal{A}[\![a_2]\!](\sigma)\}, \end{aligned}$$

where $\mathcal{A}[\![a]\!] : \mathsf{State} \to \mathbb{Z}$ denotes arithmetic evaluation. Sequential composition, conditionals, and while loops in IMP are modeled by their GKAT counterparts; **skip** is modeled by 1. Thus, IMP equivalence refines GKAT equivalence (Theorem 2.4). For example, the program transformation

$$\begin{aligned} & \mathbf{if}\ x < 0\ \mathbf{then}\ (x := 0 - x; x := 2 \times x)\ \mathbf{else}\ (x := 2 \times x) \\ \rightsquigarrow\ & (\mathbf{if}\ x < 0\ \mathbf{then}\ x := 0 - x\ \mathbf{else}\ \mathbf{skip}); x := 2 \times x \end{aligned}$$

is sound by the equivalence $pq +_b q \equiv (p +_b 1) \cdot q$. We study such equivalences further in Section 3.

2.4 Probabilistic Model

In this subsection, we give a third interpretation of GKAT expressions in terms of sub-Markov kernels, a common model for probabilistic programming languages (PPLs). We show that the language model is sound and complete for this model as well.

We briefly review some basic primitives commonly used in the denotational semantics of PPLs. For a countable set[2] X, we let $\mathcal{D}(X)$ denote the set of subdistributions over X, *i.e.*, the set of probability assignments $f : X \to [0, 1]$ summing up to at most 1—*i.e.*, $\sum_{x \in X} f(x) \leq 1$. A common distribution is the *Dirac distribution* or *point mass* on $x \in X$, denoted $\delta_x \in \mathcal{D}(X)$; it is the map $y \mapsto [y = x]$ assigning probability 1 to x, and probability 0 to $y \neq x$. (The *Iverson bracket* $[\varphi]$ is defined to be 1 if the statement φ is true, and 0 otherwise.) Denotational models of PPLs typically interpret programs as *Markov kernels*, maps of type $X \to \mathcal{D}(X)$. Such kernels can be composed in sequence using Kleisli composition, since $\mathcal{D}(-)$ is a monad [Giry 1982].

Definition 2.6 (Probabilistic Interpretation). Let $i = (\mathsf{State}, \mathrm{eval}, \mathrm{sat})$ be a triple consisting of

- a countable set of *states* State;
- for each action $p \in \Sigma$, a sub-Markov kernel $\mathrm{eval}(p) \colon \mathsf{State} \to \mathcal{D}(\mathsf{State})$; and
- for each primitive test $t \in T$, a set of states $\mathrm{sat}(t) \subseteq \mathsf{State}$.

[1] Technically, we can only reserve a test for a *finite subset* of comparisons, as T is finite. However, for reasoning about pairwise equivalences of programs, which only contain a finite number of comparisons, this restriction is not essential.
[2] We restrict to countable state spaces (i.e., discrete distributions) for ease of presentation, but this assumption is not essential. See the extend version [Smolka et al. 2019a] for a more general version using measure theory and Lebesgue integration.

Then the *probabilistic interpretation* of an expression e with respect to i is the sub-Markov kernel $\mathcal{P}_i[\![e]\!]\colon \mathrm{State} \to \mathcal{D}(\mathrm{State})$ defined as follows:

$$\mathcal{P}_i[\![p]\!] \coloneqq \mathrm{eval}(p) \qquad \mathcal{P}_i[\![b]\!](\sigma) \coloneqq [\sigma \in \mathrm{sat}^\dagger(b)] \cdot \delta_\sigma$$
$$\mathcal{P}_i[\![e \cdot f]\!](\sigma)(\sigma') \coloneqq \sum_{\sigma''} \mathcal{P}_i[\![e]\!](\sigma)(\sigma'') \cdot \mathcal{P}_i[\![f]\!](\sigma'')(\sigma')$$
$$\mathcal{P}_i[\![e +_b f]\!](\sigma) \coloneqq [\sigma \in \mathrm{sat}^\dagger(b)] \cdot \mathcal{P}_i[\![e]\!](\sigma) + [\sigma \in \mathrm{sat}^\dagger(\overline{b})] \cdot \mathcal{P}_i[\![f]\!](\sigma)$$
$$\mathcal{P}_i[\![e^{(b)}]\!](\sigma)(\sigma') \coloneqq \lim_{n\to\infty} \mathcal{P}_i[\![(e +_b 1)^n \cdot \overline{b}]\!](\sigma)(\sigma')$$

The proofs that the limit exists and that $\mathcal{P}_i[\![e]\!]$ is sub-Markov for all e can be found in the extended version of the paper [Smolka et al. 2019a].

Theorem 2.7. *The language model is sound and complete for the probabilistic model:*

$$[\![e]\!] = [\![f]\!] \quad \Longleftrightarrow \quad \forall i.\, \mathcal{P}_i[\![e]\!] = \mathcal{P}_i[\![f]\!]$$

Proof Sketch. By mutual implication.

$\Rightarrow$: For soundness, we define a map $\kappa_i\colon \mathrm{GS} \to \mathrm{State} \to \mathcal{D}(\mathrm{State})$ from guarded strings to sub-Markov kernels:

$$\kappa_i(\alpha)(\sigma) \coloneqq [\sigma \in \mathrm{sat}^\dagger(\alpha)] \cdot \delta_\sigma$$
$$\kappa_i(\alpha p w)(\sigma)(\sigma') \coloneqq [\sigma \in \mathrm{sat}^\dagger(\alpha)] \cdot \sum_{\sigma''} \mathrm{eval}(p)(\sigma)(\sigma'') \cdot \kappa_i(w)(\sigma'')(\sigma)$$

We then lift κ_i to languages via pointwise summation, $\kappa_i(L) \coloneqq \sum_{w\in L} \kappa_i(w)$, and establish that any probabilistic interpretation factors through the language model via $\kappa_i\colon \mathcal{P}_i[\![-]\!] = \kappa_i \circ [\![-]\!]$.

$\Leftarrow$: For completeness, we construct an interpretation $i \coloneqq (\mathrm{GS}, \mathrm{eval}, \mathrm{sat})$ over GS as follows,

$$\mathrm{eval}(p)(w) \coloneqq \mathrm{Unif}(\{wp\alpha \mid \alpha \in \mathrm{At}\}) \qquad \mathrm{sat}(t) \coloneqq \{x\alpha \in \mathrm{GS} \mid \alpha \le t\}$$

and show that $[\![e]\!]$ is fully determined by $\mathcal{P}_i[\![e]\!]$:

$$[\![e]\!] = \{\alpha x \in \mathrm{GS} \mid \mathcal{P}_i[\![e]\!](\alpha)(\alpha x) \neq 0\}. \qquad \square$$

As for Theorem 2.4, Theorem 2.7 can also be shown for refinement (*i.e.*, with $\subseteq$ and $\le$ instead of $=$).

Example 2.8 (Probabilistic IMP). We can extend IMP from Example 2.5 with a *probabilistic assignment* command $x \sim \mu$, where μ ranges over sub-distributions on $\mathbb{Z}$, as follows:

$$c ::= \ldots \mid x \sim \mu \qquad \Sigma = \ldots \cup \{x \sim \mu \mid x \in \mathrm{Var}, \mu \in \mathcal{D}(\mathbb{Z})\}$$

The interpretation $i = (\mathrm{Var} \to \mathbb{Z}, \mathrm{eval}, \mathrm{sat})$ is as before, except we now restrict to a finite set of variables to guarantee that the state space is countable, and interpret actions as sub-Markov kernels:

$$\mathrm{eval}(x \coloneqq n)(\sigma) \coloneqq \delta_{\sigma[x:=n]} \qquad \mathrm{eval}(x \sim \mu)(\sigma) \coloneqq \sum_{n\in\mathbb{Z}} \mu(n) \cdot \delta_{\sigma[x:=n]}$$

A concrete example of a PPL based on GKAT is McNetKAT [Smolka et al. 2019b], a recent language and verification tool for reasoning about the packet-forwarding behavior in networks.

Guarded Union Axioms

U1.	$e +_b e \equiv e$	(idempotence)
U2.	$e +_b f \equiv f +_{\overline{b}} e$	(skew commut.)
U3.	$(e +_b f) +_c g \equiv e +_{bc} (f +_c g)$	(skew assoc.)
U4.	$e +_b f \equiv be +_b f$	(guardedness)
U5.	$eg +_b fg \equiv (e +_b f) \cdot g$	(right distrib.)

Sequence Axioms (inherited from KA)

S1.	$(e \cdot f) \cdot g \equiv e \cdot (f \cdot g)$	(associativity)
S2.	$0 \cdot e \equiv 0$	(absorbing left)
S3.	$e \cdot 0 \equiv 0$	(absorbing right)
S4.	$1 \cdot e \equiv e$	(neutral left)
S5.	$e \cdot 1 \equiv e$	(neutral right)

Guarded Loop Axioms

W1.	$e^{(b)} \equiv ee^{(b)} +_b 1$	(unrolling)
W2.	$(e +_c 1)^{(b)} \equiv (ce)^{(b)}$	(tightening)
W3.	$\dfrac{g \equiv eg +_b f}{g \equiv e^{(b)} f}$ if $E(e) \equiv 0$	(fixpoint)

Fig. 1. Axioms for GKAT-expressions.

3 AXIOMATIZATION

In most programming languages, the same behavior can be realized using different programs. For example, we expect the programs **if** b **then** e **else** f and **if** (**not** b) **then** f **else** e to encode the same behavior. Likewise, different expressions in GKAT can denote the same language of guarded strings. For instance, the previous example is reflected in GKAT by the fact that the language semantics of $e +_b f$ and $f +_{\overline{b}} e$ coincide. This raises the questions: what other equivalences hold between GKAT expressions? And, can all equivalences be captured by a finite number of equations? In this section, we give some initial answers to these questions, by proposing a set of axioms for GKAT and showing that they can be used to prove a large class of equivalences.

3.1 Some Simple Axioms

As an initial answer to the first question, we propose the following.

Definition 3.1. We define $\equiv$ as the smallest congruence (with respect to all operators) on Exp that satisfies the axioms given in Figure 1 (for all $e, f, g \in \mathrm{Exp}$ and $b, c, d \in \mathrm{BExp}$) and subsumes Boolean equivalence in the sense that $b \equiv_{\mathrm{BA}} c$ implies $b \equiv c$.

The guarded union axioms (U1-U5) can be understood intuitively in terms of conditionals. For instance, we have the law $e +_b f \equiv f +_{\overline{b}} e$ discussed before, but also $eg +_b fg \equiv (e +_b f) \cdot g$, which says that g can be "factored out" of branches of a guarded union. Equivalences for sequential composition are also intuitive. For instance, $0 \cdot e \equiv 0$ encodes that any instruction after failure is irrelevant, because the program has failed. The axioms for loops (W1–W3) are more subtle. The axiom $e^{(b)} \equiv ee^{(b)} +_b 1$ (W1) says that we can think of a guarded loop as equivalent to its unrolling—*i.e.*, the program **while** b **do** e has the same behavior as the program **if** b **then** (e; **while** b **do** e) **else skip**. The axiom $(e +_c 1)^{(b)} \equiv (ce)^{(b)}$ (W2) states that if part of a loop body does not have an effect (*i.e.*, is equivalent to **skip**), it can be omitted; we refer to this transformation as *loop tightening*.

To explain the fixpoint axiom (W3), disregard the side-condition for a moment. In a sense, this rule states that if g tests (using b) whether to execute e and loop again or execute f (*i.e.*, if $g \equiv eg +_b f$) then g is a b-guarded loop followed by f (*i.e.*, $g \equiv e^{(b)} f$). However, such a rule is not sound in general. For instance, suppose $e, f, g, b = 1$; in that case, $1 \equiv 1 \cdot 1 +_1 1$ can be proved using the other axioms, but applying the rule would allow us to conclude that $1 \equiv 1^{(1)} \cdot 1$, even though $[\![1]\!] = \mathrm{At}$ and $[\![1^{(1)} \cdot 1]\!] = \varnothing$! The problem here is that, while g is tail-recursive as required by the premise, this self-similarity is trivial because e does not represent a productive program. We thus need to restrict the application of the inference rule to cases where the loop body is *strictly productive*—*i.e.*, where e is guaranteed to execute *some* action. To this end, we define the function E as follows.

Guarded Union Facts

U3'.	$e +_b (f +_c g) \equiv (e +_b f) +_{b+c} g$	(skew assoc.)
U4'.	$e +_b f \equiv e +_b \overline{b} f$	(guardedness)
U5'.	$b \cdot (e +_c f) \equiv be +_c bf$	(left distrib.)
U6.	$e +_b 0 \equiv be$	(neutral right)
U7.	$e +_0 f \equiv f$	(trivial right)
U8.	$b \cdot (e +_b f) \equiv be$	(branch selection)

Guarded Iteration Facts

W4.	$e^{(b)} \equiv e^{(b)} \cdot \overline{b}$	(guardedness)
W4'.	$e^{(b)} \equiv (be)^{(b)}$	(guardedness)
W5.	$e^{(0)} \equiv 1$	(neutrality)
W6.	$e^{(1)} \equiv 0$	(absorption)
W6'.	$b^{(c)} \equiv \overline{c}$	(absorption)
W7.	$e^{(c)} \equiv e^{(bc)} \cdot e^{(c)}$	(fusion)

Fig. 2. Derivable GKAT facts

Definition 3.2. The function $E : \mathsf{Exp} \to \mathsf{BExp}$ is defined inductively as follows:

$$E(b) \coloneqq b \qquad E(p) \coloneqq 0 \qquad E(e +_b f) \coloneqq b \cdot E(e) + \overline{b} \cdot E(f) \qquad E(e \cdot f) \coloneqq E(e) \cdot E(f) \qquad E(e^{(b)}) \coloneqq \overline{b}$$

Intuitively, $E(e)$ is the weakest test that guarantees that e terminates successfully, but does not perform any action. For instance, $E(p)$ is 0—the program p is guaranteed to perform the action p. Using E, we can now restrict the application of the fixpoint rule to the cases where $E(e) \equiv 0$, *i.e.*, where e performs an action under any circumstance.

Theorem 3.3 (Soundness). *The GKAT axioms are sound for the language model: for all $e, f \in \mathsf{Exp}$,*

$$e \equiv f \quad \Longrightarrow \quad [\![e]\!] = [\![f]\!].$$

Proof Sketch. By induction on the length of derivation of the congruence $\equiv$. We provide the full proof in the extended version [Smolka et al. 2019a] and show just the proof for the fixpoint rule. Here, we should argue that if $E(e) \equiv 0$ and $[\![g]\!] = [\![eg +_b f]\!]$, then also $[\![g]\!] = [\![e^{(b)} f]\!]$. We note that, using soundness of (W1) and (U5), we can derive that $[\![e^{(b)} f]\!] = [\![(ee^{(b)} +_b 1) f]\!] = [\![ee^{(b)} f +_b f]\!]$.

We reason by induction on the length of guarded strings. In the base case, we know that $\alpha \in [\![g]\!]$ if and only if $\alpha \in [\![eg +_b f]\!]$; since $E(e) \equiv 0$, the latter holds precisely when $\alpha \in [\![f]\!]$ and $\alpha \leq \overline{b}$, which is equivalent to $\alpha \in [\![e^{(b)} f]\!]$. For the inductive step, suppose the claim holds for y; then

$$\begin{aligned}
& \alpha p y \in [\![g]\!] \\
\iff & \alpha p y \in [\![eg +_b f]\!] \\
\iff & \alpha p y \in [\![eg]\!] \wedge \alpha \leq b \ \ \textbf{or} \ \ \alpha p y \in [\![f]\!] \wedge \alpha \leq \overline{b} \\
\iff & \exists y, \beta.\ y = y_1 \beta y_2 \wedge \alpha p y_1 \beta \in [\![e]\!] \wedge \beta y_2 \in [\![g]\!] \wedge \alpha \leq b \ \ \textbf{or} \ \ \alpha p y \in [\![f]\!] \wedge \alpha \leq \overline{b} && (E(e) = 0) \\
\iff & \exists y, \beta.\ y = y_1 \beta y_2 \wedge \alpha p y_1 \beta \in [\![e]\!] \wedge \beta y_2 \in [\![e^{(b)} f]\!] \wedge \alpha \leq b \ \ \textbf{or} \ \ \alpha p y \in [\![f]\!] \wedge \alpha \leq \overline{b} && (\text{IH}) \\
\iff & \alpha p y \in [\![ee^{(b)} f]\!] \wedge \alpha \leq b \ \ \textbf{or} \ \ \alpha p y \in [\![f]\!] \wedge \alpha \leq \overline{b} && (E(e) = 0) \\
\iff & \alpha p y \in [\![ee^{(b)} f +_b f]\!] = [\![e^{(b)} f]\!] && \square
\end{aligned}$$

3.2 A Fundamental Theorem

The side condition on (W3) is inconvenient when proving facts about loops. However, it turns out that we can transform any loop into an equivalent, *productive* loop—*i.e.*, one with a loop body e such that $E(e) \equiv 0$. To this end, we need a way of decomposing a GKAT expression into a guarded sum of an expression that describes termination, and another (strictly productive) expression that describes the next steps that the program may undertake. As a matter of fact, we already have a handle on the former term: $E(e)$ is a Boolean term that captures the atoms for which e may halt immediately. It therefore remains to describe the next steps of a program.

Definition 3.4 (Derivatives). For $\alpha \in \mathrm{At}$ we define $D_\alpha : \mathrm{Exp} \to 2 + \Sigma \times \mathrm{Exp}$ inductively as follows, where $2 = \{0, 1\}$ is the two-element set:

$$D_\alpha(b) = \begin{cases} 1 & \alpha \leq b \\ 0 & \alpha \not\leq b \end{cases} \qquad D_\alpha(p) = (p, 1) \qquad D_\alpha(e +_b f) = \begin{cases} D_\alpha(e) & \alpha \leq b \\ D_\alpha(f) & \alpha \leq \overline{b} \end{cases}$$

$$D_\alpha(e \cdot f) = \begin{cases} (p, e' \cdot f) & D_\alpha(e) = (p, e') \\ 0 & D_\alpha(e) = 0 \\ D_\alpha(f) & D_\alpha(e) = 1 \end{cases} \qquad D_\alpha(e^{(b)}) = \begin{cases} (p, e' \cdot e^{(b)}) & \alpha \leq b \wedge D_\alpha(e) = (p, e') \\ 0 & \alpha \leq b \wedge D_\alpha(e) \in 2 \\ 1 & \alpha \leq \overline{b} \end{cases}$$

We will use a general type of guarded union to sum over an atom-indexed set of expressions.

Definition 3.5. Let $\Phi \subseteq \mathrm{At}$, and let $\{e_\alpha\}_{\alpha \in \Phi}$ be a set of expressions indexed by Φ. We write

$$\mathop{+}_{\alpha \in \Phi} e_\alpha = \begin{cases} e_\beta +_\beta \left(\mathop{+}_{\alpha \in \Phi \setminus \{\beta\}} e_\alpha \right) & \beta \in \Phi \\ 0 & \Phi = \varnothing \end{cases}$$

Like other operators on indexed sets, we may abuse notation and replace Φ by a predicate over some atom α, with e_α a function of α; for instance, we could write $+_{\alpha \leq 1} \alpha \equiv 1$.

Remark 3.6. The definition above is ambiguous in the choice of β. However, because of skew-commutativity (U2) and skew-associativity (U3), that does not change the meaning of the expression as far as $\equiv$ is concerned. For the details, see the extended version [Smolka et al. 2019a].

We are now ready to state the desired decomposition of terms. Following [Rutten 2000; Silva 2010], we call this the *fundamental theorem* of GKAT, in reference to the strong analogy with the fundamental theorem of calculus, as explained in [Rutten 2000; Silva 2010]. The proof is included in the extended version [Smolka et al. 2019a].

Theorem 3.7 (Fundamental Theorem). *For all GKAT programs e, the following equality holds:*

$$e \equiv 1 +_{E(e)} D(e), \qquad \text{where } D(e) \coloneqq \mathop{+}_{\alpha \,:\, D_\alpha(e) = (p_\alpha, e_\alpha)} p_\alpha \cdot e_\alpha. \tag{1}$$

The following observations about D and E are also useful.

Lemma 3.8. *Let $e \in \mathrm{Exp}$; its components $E(e)$ and $D(e)$ satisfy the following identities:*

$$E(D(e)) \equiv 0 \qquad \overline{E(e)} \cdot D(e) \equiv D(e) \qquad \overline{E(e)} \cdot e \equiv D(e)$$

Using the fundamental theorem and the above, we can now show how to syntactically transform any loop into an equivalent loop whose body e is strictly productive.

Lemma 3.9 (Productive Loop). *Let $e \in \mathrm{Exp}$ and $b \in \mathrm{BExp}$. We have $e^{(b)} \equiv D(e)^{(b)}$.*

Proof. Using Lemma 3.8, we derive as follows:

$$e^{(b)} \overset{\text{FT}}{\equiv} (1 +_{E(e)} D(e))^{(b)} \overset{\text{U2}}{\equiv} (D(e) +_{\overline{E(e)}} 1)^{(b)} \overset{\text{W2}}{\equiv} (\overline{E(e)} D(e))^{(b)} \equiv D(e)^{(b)} \qquad \square$$

3.3 Derivable Facts

The GKAT axioms can be used to derive other natural equivalences of programs, such as the ones in Figure 2. For instance, $e^{(b)} \equiv e^{(b)}\overline{b}$, labelled (W4), says that b must be false when $e^{(b)}$ ends.

Lemma 3.10. *The facts in Figure 2 are derivable from the axioms.*

Proof Sketch. Let us start by showing (U6).

$$\begin{aligned}
e +_b 0 &\equiv be +_b 0 && \text{(U4. } e +_b f \equiv be +_b f\text{)}\\
&\equiv 0 +_{\overline{b}} be && \text{(U2. } e +_b f \equiv f +_{\overline{b}} e\text{)}\\
&\equiv \overline{b}be +_{\overline{b}} be && \text{(Boolean algebra and S2. } 0 \equiv 0e\text{)}\\
&\equiv be +_{\overline{b}} be && \text{(U4. } e +_b f \equiv be +_b f\text{)}\\
&\equiv be && \text{(U1. } e +_b e \equiv e\text{)}
\end{aligned}$$

To prove (W7), we use the productive loop lemma and the fixpoint axiom (W3).

$$\begin{aligned}
e^{(c)} &\equiv e^{(c)} +_{bc} e^{(c)} && \text{(U1. } e +_b e \equiv e\text{)}\\
&\equiv (D(e))^{(c)} +_{bc} e^{(c)} && \text{(Productive loop lemma)}\\
&\equiv (D(e)D(e)^{(c)} +_c 1) +_{bc} e^{(c)} && \text{(W1. } e^{(b)} \equiv ee^{(b)} +_b 1\text{)}\\
&\equiv c \cdot (D(e)D(e)^{(c)} +_c 1) +_{bc} e^{(c)} && \text{(U4 and Boolean algebra)}\\
&\equiv c \cdot D(e)D(e)^{(c)} +_{bc} e^{(c)} && \text{(U8. } b \cdot (e +_b f) \equiv be\text{)}\\
&\equiv D(e)D(e)^{(c)} +_{bc} e^{(c)} && \text{(U4 and Boolean algebra)}\\
&\equiv D(e)e^{(c)} +_{bc} e^{(c)} && \text{(Productive loop lemma)}\\
&\equiv D(e)^{(bc)}e^{(c)} && \text{(W3)}\\
&\equiv e^{(bc)}e^{(c)} && \text{(Productive loop lemma)}
\end{aligned}$$

The remaining proofs appear in the extended version [Smolka et al. 2019a]. □

We conclude our presentation of derivable facts by showing one more interesting fact. Unlike the derived facts above, this one is an implication: if the test c is invariant for the program e given that a test b succeeds, then c is preserved by a b-loop on e.

Lemma 3.11 (Invariance). *Let $e \in \mathsf{Exp}$ and $b, c \in \mathsf{BExp}$. If $cbe \equiv cbec$, then $ce^{(b)} \equiv (ce)^{(b)}c$.*

Proof. We first derive a useful equivalence, as follows:

$$\begin{aligned}
cb \cdot D(e) &\equiv cb \cdot \overline{E(e)} \cdot e && \text{(Lemma 3.8)}\\
&\equiv \overline{E(e)} \cdot cbe && \text{(Boolean algebra)}\\
&\equiv \overline{E(e)} \cdot cbec && \text{(premise)}\\
&\equiv cb \cdot \overline{E(e)} \cdot ec && \text{(Boolean algebra)}\\
&\equiv cb \cdot D(e) \cdot c && \text{(Lemma 3.8)}
\end{aligned}$$

Next, we show the main claim by deriving

$$\begin{aligned}
ce^{(b)} &\equiv c \cdot D(e)^{(b)} && \text{(Productive loop lemma)}\\
&\equiv c \cdot (D(e) \cdot D(e)^{(b)} +_b 1) && \text{(W1)}\\
&\equiv c \cdot (D(e) \cdot e^{(b)} +_b 1) && \text{(Productive loop lemma)}\\
&\equiv c \cdot (b \cdot D(e) \cdot e^{(b)} +_b 1) && \text{(U2)}\\
&\equiv cb \cdot D(e) \cdot e^{(b)} +_b c && \text{(U5')}\\
&\equiv cb \cdot D(e) \cdot ce^{(b)} +_b c && \text{(above derivation)}
\end{aligned}$$

$$\equiv c \cdot D(e) \cdot ce^{(b)} +_b c \qquad \text{(U2)}$$
$$\equiv (c \cdot D(e))^{(b)} c \qquad \text{(W3)}$$
$$\equiv D(ce)^{(b)} c \qquad \text{(Def. } D\text{, Boolean algebra)}$$
$$\equiv (ce)^{(b)} c \qquad \text{(Productive loop lemma)}$$

This completes the proof. □

3.4 A Limited Form of Completeness

Above, we considered a number of axioms that were proven sound with respect to the language model. Ultimately, we would like to show the converse, *i.e.*, that these axioms are sufficient to prove all equivalences between programs, meaning that whenever $[\![e]\!] = [\![f]\!]$, it also holds that $e \equiv f$.

We return to this general form of completeness in Section 6, when we can rely on the coalgebraic theory of GKAT developed in Sections 4 and 5. At this point, however, we can already prove a special case of completeness related to Hoare triples. Suppose e is a GKAT program, and b and c are Boolean expressions encoding pre- and postconditions. The equation $[\![be]\!] = [\![bec]\!]$ states that every finite, terminating run of e starting from a state satisfying b concludes in a state satisfying c. The following states that all valid Hoare triples of this kind can be established axiomatically:

Theorem 3.12 (Hoare completeness). *Let $e \in \mathsf{Exp}$, $b, c \in \mathsf{BExp}$. If $[\![bec]\!] = [\![be]\!]$, then $bec \equiv be$.*

Proof Sketch. By induction on e. We show only the case for while loops and defer the full proof to the extended version [Smolka et al. 2019a].

If $e = e_0^{(d)}$, first note that if $b \equiv 0$, then the claim follows trivially. For $b \not\equiv 0$, let

$$h = \sum \{\alpha \in \mathsf{At} : \exists n. [\![b]\!] \diamond [\![de_0]\!]^n \diamond [\![\alpha]\!] \neq \varnothing\}.$$

We make the following observations.

(i) Since $b \not\equiv 0$, we have that $[\![b]\!] \diamond [\![de_0]\!]^0 \diamond [\![b]\!] = [\![b]\!] \neq \varnothing$, and thus $b \leq h$.

(ii) If $\alpha \leq h\overline{d}$, then in particular $\gamma w \alpha \in [\![b]\!] \diamond [\![de_0]\!]^n \diamond [\![\alpha]\!]$ for some n and γw. Since $\alpha \leq \overline{d}$, it follows that $\gamma w \alpha \in [\![be_0^{(d)}]\!] = [\![be_0^{(d)} c]\!]$, and thus $\alpha \leq c$. Consequently, $h\overline{d} \leq c$.

(iii) If $\alpha w \beta \in [\![dhe_0]\!]$, then $\alpha \leq h$ and hence there exists an n such that $\gamma x \alpha \in [\![b]\!] \diamond [\![de_0]\!]^n \diamond [\![\beta]\!]$. But then $\gamma x \alpha w \beta \in [\![b]\!] \diamond [\![de_0]\!]^{n+1} \diamond [\![\beta]\!]$, and therefore $\beta \leq h$. We can conclude that $[\![dhe_0]\!] = [\![dhe_0 h]\!]$; by induction, it follows that $dhe_0 h \equiv dhe_0$.

Using these observations and the invariance lemma (Lemma 3.11), we derive

$$be_0^{(d)} c \equiv bhe_0^{(d)} c \qquad \text{(By (i))}$$
$$\equiv b \cdot (he_0)^{(d)} hc \qquad \text{(Invariance and (iii))}$$
$$\equiv b \cdot (he_0)^{(d)} \overline{d} hc \qquad \text{(W4)}$$
$$\equiv b \cdot (he_0)^{(d)} \overline{d} h \qquad \text{(By (ii))}$$
$$\equiv b \cdot (he_0)^{(d)} h \qquad \text{(W4)}$$
$$\equiv bhe_0{}^{(d)} \qquad \text{(Invariance and (iii))}$$
$$\equiv be_0{}^{(d)} \qquad \text{(By (i))}$$

This completes the proof. □

As a special case, the fact that a program has no traces at all can be shown axiomatically.

Corollary 3.13 (Partial Completeness). *If $[\![e]\!] = \varnothing$, then $e \equiv 0$.*

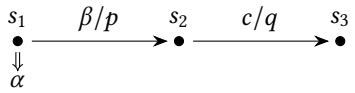


Fig. 3. Graphical depiction of a G-coalgebra $\langle X, \delta^{\mathcal{X}} \rangle$. States are represented by dots, labeled with the name of that state whenever relevant. In this example, $\delta^{\mathcal{X}}(s_1)(\alpha) = 1$, and $\delta^{\mathcal{X}}(s_1)(\beta) = (p, s_2)$. When $\gamma \in \mathsf{At}$ such that $\delta^{\mathcal{X}}(s)(\gamma) = 0$, we draw no edge at all. We may abbreviate drawings by combining transitions with the same target into a Boolean expression; for instance, when $c = \alpha + \beta$, we have $\delta^{\mathcal{X}}(s_2)(\alpha) = \delta^{\mathcal{X}}(s_2)(\beta) = (q, s_3)$.

Proof. We have $[\![1 \cdot e]\!] = [\![e]\!] = \varnothing = [\![1 \cdot e \cdot 0]\!]$, and thus $e \equiv 1 \cdot e \equiv 1 \cdot e \cdot 0 \equiv 0$ by Theorem 3.12. □

We will return to deriving a general completeness result in Section 6. This will rely on the coalgebraic theory of GKAT, which we develop next (Sections 4 and 5).

4 AUTOMATON MODEL AND KLEENE THEOREM

In this section, we present an automaton model that accepts traces (*i.e.*, guarded strings) of GKAT programs. We then present language-preserving constructions from GKAT expressions to automata, and conversely, from automata to expressions. Our automaton model is rich enough to express programs that go beyond GKAT; in particular, it can encode traces of programs with **goto** statements that have no equivalent GKAT program [Kozen and Tseng 2008]. In order to obtain a Kleene Theorem for GKAT, that is, a correspondence between automata and GKAT programs, we identify conditions ensuring that the language accepted by an automaton corresponds to a valid GKAT program.

4.1 Automata and Languages

Let G be the functor $GX = (2 + \Sigma \times X)^{\mathsf{At}}$, where $2 = \{0, 1\}$ is the two-element set. A *G-coalgebra* is a pair $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ with *state space* X and *transition map* $\delta^{\mathcal{X}} : X \to GX$. The outcomes 1 and 0 model immediate acceptance and rejection, respectively. From each state $s \in X$, given an input $\alpha \in \mathsf{At}$, the coalgebra performs exactly one of three possible actions: it either produces an output $p \in \Sigma$ and moves to a new state t, halts and accepts, or halts and rejects; that is, either $\delta^{\mathcal{X}}(s)(\alpha) = (p, t)$, or $\delta^{\mathcal{X}}(s)(\alpha) = 1$, or $\delta^{\mathcal{X}}(s)(\alpha) = 0$.

A *G-automaton* is a G-coalgebra with a designated start state ι, commonly denoted as a triple $\mathcal{X} = \langle X, \delta^{\mathcal{X}}, \iota \rangle$. We can represent G-coalgebras graphically as in Figure 3.

A G-coalgebra $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ can be viewed both as an acceptor of finite guarded strings $\mathsf{GS} = \mathsf{At} \cdot (\Sigma \cdot \mathsf{At})^*$, or as an acceptor of finite *and* infinite guarded strings $\mathsf{GS} \cup \omega\text{-}\mathsf{GS}$, where $\omega\text{-}\mathsf{GS} \coloneqq (\mathsf{At} \cdot \Sigma)^\omega$. Acceptance for a state s is captured by the following equivalences:

$$\begin{aligned} \mathsf{accept}(s, \alpha) &\iff \delta^{\mathcal{X}}(s)(\alpha) = 1 \\ \mathsf{accept}(s, \alpha p x) &\iff \exists t.\ \delta^{\mathcal{X}}(s)(\alpha) = (p, t) \wedge \mathsf{accept}(t, x) \end{aligned} \tag{2}$$

The language of finite guarded strings $\ell^{\mathcal{X}}(s) \subseteq GS$ accepted from state $s \in X$ is the *least fixpoint* solution of the above system; in other words, we interpret (2) inductively. The language of finite and infinite guarded strings $L^{\mathcal{X}}(s) \subseteq \mathsf{GS} \cup \omega\text{-}\mathsf{GS}$ accepted from state s is the *greatest fixpoint* solution of the above system; in other words, we interpret (2) coinductively.[3] The two languages are related by the equation $\ell^{\mathcal{X}}(s) = L^{\mathcal{X}}(s) \cap \mathsf{GS}$. Our focus will mostly be on the finite-string semantics, $\ell^{\mathcal{X}}(-) : X \to 2^{\mathsf{GS}}$, since GKAT expressions denote finite-string languages, $[\![-]\!] : \mathsf{Exp} \to 2^{\mathsf{GS}}$.

[3]The set $\mathcal{F}$ of maps $F : X \to 2^{\mathsf{GS} \cup \omega\text{-}\mathsf{GS}}$ ordered pointwise by subset inclusion forms a complete lattice. The monotone map

$$\tau : \mathcal{F} \to \mathcal{F},\ \tau(F) = \lambda s \in X.\ \{\alpha \in \mathsf{At} \mid \delta^{\mathcal{X}}(s)(\alpha) = 1\} \cup \{apx \mid \exists t.\ \delta^{\mathcal{X}}(s)(\alpha) = (p, t) \wedge x \in F(t)\}$$

arising from (2) has least and greatest fixpoints, $\ell^{\mathcal{X}}$ and $L^{\mathcal{X}}$, by the Knaster-Tarksi theorem.

e	X_e	$\delta_e \in X_e \to GX_e$	$\iota_e(\alpha) \in 2 + \Sigma \times X_e$
b	$\varnothing$	$\varnothing$	$[\alpha \leq b]$
p	$\{*\}$	$* \mapsto \mathbf{1}$	$(p, *)$
$f +_b g$	$X_f + X_g$	$\delta_f + \delta_g$	$\begin{cases} \iota_f(\alpha) & \alpha \leq b \\ \iota_g(\alpha) & \alpha \leq \overline{b} \end{cases}$
$f \cdot g$	$X_f + X_g$	$(\delta_f + \delta_g)[X_f, \iota_g]$	$\begin{cases} \iota_f(\alpha) & \iota_f(\alpha) \neq 1 \\ \iota_g(\alpha) & \iota_f(\alpha) = 1 \end{cases}$
$f^{(b)}$	X_f	$\delta_f[X_f, \iota_e]$	$\begin{cases} 1 & \alpha \leq \overline{b} \\ 0 & \alpha \leq b, \iota_f(\alpha) = 1 \\ \iota_f(\alpha) & \alpha \leq b, \iota_f(\alpha) \neq 1 \end{cases}$

Fig. 4. Construction of the Thompson coalgebra $\mathcal{X}_e = \langle X_e, \delta_e \rangle$ with initial pseudostate ι_e.

The language accepted by a G-automaton $\mathcal{X} = \langle X, \delta^X, \iota \rangle$ is the language accepted by its initial state ι. Just like the language model for GKAT programs, the language semantics of a G-automaton satisfies the determinacy property (see Definition 2.2). In fact, every language that satisfies the determinacy property can be recognized by a G-automaton, possibly with infinitely many states. (We will prove this formally in Theorem 5.8.)

4.2 Expressions to Automata: a Thompson Construction

We translate expressions to G-coalgebras using a construction reminiscent of Thompson's construction for regular expressions [Thompson 1968], where automata are formed by induction on the structure of the expressions and combined to reflect the various GKAT operations.

We first set some notation. A *pseudostate* is an element $h \in GX$. We let $\mathbf{1} \in GX$ denote the pseudostate $\mathbf{1}(\alpha) = 1$, *i.e.*, the constant function returning 1. Let $\mathcal{X} = \langle X, \delta \rangle$ be a G-coalgebra. The *uniform continuation* of $Y \subseteq X$ by $h \in GX$ (in $\mathcal{X}$) is the coalgebra $\mathcal{X}[Y, h] := \langle X, \delta[Y, h] \rangle$, where

$$\delta[Y, h](x)(\alpha) := \begin{cases} h(\alpha) & \text{if } x \in Y, \delta(x)(\alpha) = 1 \\ \delta(x)(\alpha) & \text{otherwise.} \end{cases}$$

Intuitively, uniform continuation replaces termination of states in a region Y of $\mathcal{X}$ by a transition described by $h \in GX$; this construction will be useful for modeling operations that perform some kind of sequencing. Figure 5 schematically describes the uniform continuation operation, illustrating different changes to the automaton that can occur as a result; observe that since h may have transitions into Y, uniform continuation can introduce loops.

We will also need coproducts to combine coalgebras. Intuitively, the coproduct of two coalgebras is just the juxtaposition of both coalgebras. Formally, for $\mathcal{X} = \langle X, \delta_1 \rangle$ and $\mathcal{Y} = \langle Y, \delta_2 \rangle$, we write the coproduct as $\mathcal{X} + \mathcal{Y} = \langle X + Y, \delta_1 + \delta_2 \rangle$, where $X + Y$ is the disjoint union of X and Y, and $\delta_1 + \delta_2 \colon X + Y \to G(X + Y)$ is the map that applies δ_1 to states in X and δ_2 to states in Y.

Figure 4 presents our translation from expressions e to coalgebras $\mathcal{X}_e$ using coproducts and uniform continuations, and Figure 6 sketches the transformations used to construct the automaton of a term from its subterms. We model initial states as pseudostates, rather than proper states. This trick avoids the ε-transitions that appear in the classical Thompson construction and yields compact, linear-size automata. Figure 7 depicts some examples of our construction.

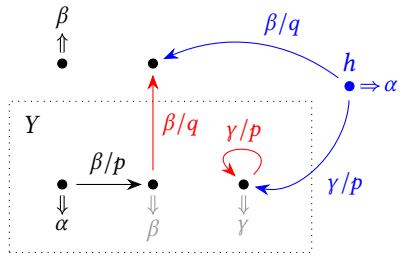
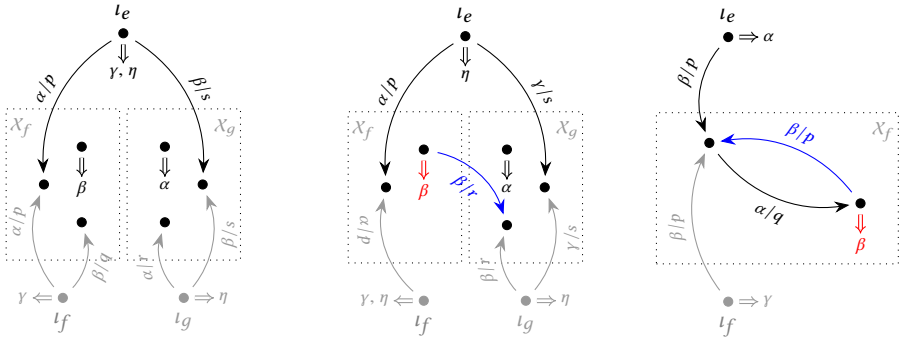


Fig. 5. Schematic explanation of the *uniform continuation* $\mathcal{X}[Y, h]$ of $\mathcal{X}$, where $Y \subseteq X$ and $h \in GX$. The pseudostate h and its transitions are drawn in blue. Transitions present in $\mathcal{X}$ unchanged by the extension are drawn in black; grayed out transitions are replaced by transitions drawn in red as a result of the extension.



(a) $e = f +_b g$, with $\alpha, \gamma \leq b$ and $\beta, \eta \leq \overline{b}$. (b) $e = f \cdot g$ (c) $e = f^{(b)}$, with $\beta, \gamma \leq b$ and $\alpha \leq \overline{b}$

Fig. 6. Schematic depiction of the Thompson construction for guarded union, sequencing and guarded loop operators. The initial psuedostates of the automata for f and g are depicted in gray. Transitions in red are present in the automata for f and g, but overridden by a uniform extension with the transitions in blue.

To turn the resulting coalgebra into an automaton, we simply convert the initial pseudostate into a proper state. Formally, when $\mathcal{X}_e = \langle X_e, \delta_e \rangle$, we write $\mathcal{X}_e^\iota$ for the G-automaton $\langle \{\iota\} + X_e, \delta_e^\iota, \iota \rangle$, where for $x \in X_e$, we set $\delta_e^\iota(x) = \delta_e(x)$ as well as $\delta_e^\iota(\iota) = \iota_e$. We call $\mathcal{X}_e$ and $\mathcal{X}_e^\iota$ the *Thompson coalgebra* and *Thompson automaton* for e, respectively.
The construction translates expressions to equivalent automata in the following sense:

Theorem 4.1 (Correctness I). *The Thompson automaton for e recognizes $[\![e]\!]$, that is $\ell^{\mathcal{X}_e^\iota}(\iota) = [\![e]\!]$.*

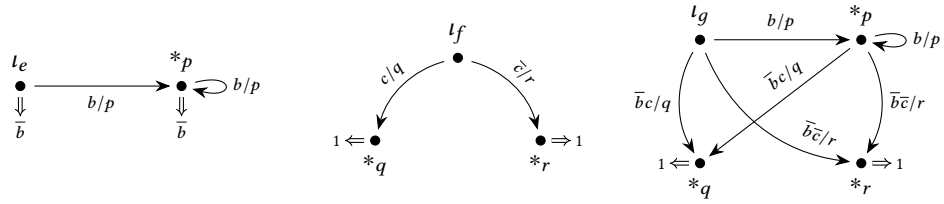
Proof Sketch. This is a direct corollary of Proposition 4.5 and Theorem 4.8, to follow. □

Moreover, the construction is efficiently implementable and yields small automata:

Proposition 4.2. *The Thompson automaton for e is effectively constructible in time $O(|e|)$ and has $\#_\Sigma(e) + 1$ (thus, $O(|e|)$) states, where $|\mathsf{At}|$ is considered a constant for the time complexity claim, $|e|$ denotes the size of the expression, and $\#_\Sigma(e)$ denotes the number of occurrences of actions in e.*

4.3 Automata to Expressions: Solving Linear Systems

The previous construction shows that every GKAT expression can be translated to an equivalent G-automaton. In this section we consider the reverse direction, from G-automata to GKAT expressions.



(a) $e = \textbf{while}\ b\ \textbf{do}\ p$ (b) $f = \textbf{if}\ c\ \textbf{then}\ q\ \textbf{else}\ r$ (c) $g = e \cdot f$

Fig. 7. Concrete construction of an automaton using the Thompson construction. First, we construct an automaton for e, then an automaton for f, and finally we combine these into an automaton for g. In these examples, p, q, r are single action letters, not arbitrary expressions.

The main idea is to interpret the coalgebra structure as a system of equations, with one variable and equation per state, and show that there are GKAT expressions solving the system, modulo equivalence; this idea goes back to Conway [1971] and Backhouse [1975]. Not all systems arising from G-coalgebras have a solution, and so not all G-coalgebras can be captured by GKAT expressions. However, we identify a subclass of G-coalgebras that can be represented as GKAT terms. By showing that this class contains the coalgebras produced by our expressions-to-automata translation, we obtain an equivalence between GKAT expressions and coalgebras in this class.

We start by defining when a map assigning expressions to coalgebra states is a solution.

Definition 4.3 (Solution). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ be a G-coalgebra. We say that $s \colon X \to \mathrm{Exp}$ is a *solution* to $\mathcal{X}$ if for all $x \in X$ it holds that

$$s(x) \equiv \sum_{\alpha \leq 1} \lfloor \delta^{\mathcal{X}}(x)(\alpha) \rfloor_s \qquad \text{where} \quad \lfloor 0 \rfloor_s := 0 \quad \lfloor 1 \rfloor_s := 1 \quad \lfloor \langle p, x \rangle \rfloor_s := p \cdot s(x)$$

Example 4.4. Consider the Thompson automata in Figure 7.

(a) Solving the first automaton requires, by Definition 4.3, finding an expression $s_e(*_p)$ such that $s_e(*_p) \equiv p \cdot s_e(*_p) +_b 1$. By (W1), we know that $s_e(*_p) = p^{(b)}$ is valid; in fact, (W3) tells us that this choice of x is the *only* valid solution up to GKAT-equivalence. If we include ι_e as a state, we can choose $s_e(\iota_e) = p^{(b)}$ as well.

(b) The second automaton has an easy solution: both $*_q$ and $*_r$ are solved by setting $s_f(*_q) = s_f(*_r) = 1$. If we include ι_f as a state, we can choose $s_f(\iota_f) = q \cdot s_f(*_q) +_b r \cdot s_f(*_r) \equiv q +_b r$.

(c) The third automaton was constructed from the first two; similarly, we can construct its solution from the solutions to the first two. We set $s_g(*_p) = s_e(*_p) \cdot s_f(\iota_f)$, and $s_g(*_q) = s_f(*_q)$, and $s_g(*_r) = s_f(*_r)$. If we include ι_g as a state, we can choose $s_g(\iota_g) = s_e(\iota_e) \cdot s_f(\iota_f)$.

Solutions are language-preserving maps from states to expressions in the following sense:

Proposition 4.5. *If s solves $\mathcal{X}$ and x is a state, then $[\![s(x)]\!] = \ell^{\mathcal{X}}(x)$.*

Proof Sketch. Show that $w \in [\![s(x)]\!] \Leftrightarrow w \in \ell^{\mathcal{X}}(x)$ by induction on the length of $w \in \mathrm{GS}$. □

We would like to build solutions for G-coalgebras, but Kozen and Tseng [2008] showed that this is not possible in general: there is a 3-state G-coalgebra that does not correspond to any **while** program, but instead can only be modeled by a program with multi-level breaks. In order to obtain an exact correspondence to GKAT programs, we first identify a sufficient condition for G-coalgebras to permit solutions, and then show that the Thompson coalgebra defined previously meets this condition.

Definition 4.6 (Well-nested Coalgebra). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ and $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}} \rangle$ range over G-coalgebras. The collection of *well-nested* coalgebras is inductively defined as follows:

(i) If $\mathcal{X}$ has no transitions, *i.e.*, if $\delta^{\mathcal{X}} \in X \to 2^{\mathsf{At}}$, then $\mathcal{X}$ is well-nested.
(ii) If $\mathcal{X}$ and $\mathcal{Y}$ are well-nested and $h \in G(X + Y)$, then $(\mathcal{X} + \mathcal{Y})[X, h]$ is well-nested.

We are now ready to construct solutions to well-nested coalgebras.

Theorem 4.7 (Existence of Solutions). *Any well-nested coalgebra admits a solution.*

Proof Sketch. Assume $\mathcal{X}$ is well-nested. We proceed by rule induction on the well-nestedness derivation.

(i) Suppose $\delta^{\mathcal{X}} : X \to 2^{\mathsf{At}}$. Then

$$s^{\mathcal{X}}(x) := \sum \{\alpha \in \mathsf{At} \mid \delta^{\mathcal{X}}(x)(\alpha) = 1\}$$

is a solution to $\mathcal{X}$.

(ii) Let $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}} \rangle$ and $\mathcal{Z} = \langle Z, \delta^{\mathcal{Z}} \rangle$ be well-nested G-coalgebras, and let $h \in G(Y + Z)$ be such that $\mathcal{X} = (\mathcal{Y} + \mathcal{Z})[Y, h]$. By induction, $\mathcal{Y}$ and $\mathcal{Z}$ admit solutions $s^{\mathcal{Y}}$ and $s^{\mathcal{Z}}$ respectively; we need to find a solution $s^{\mathcal{X}}$ to $X = Y + Z$. The idea is to retain the solution that we had for states in $\mathcal{Z}$—whose behavior has not changed under uniform continuation—while modifying the solution to states in $\mathcal{Y}$ in order to account for transitions from h. To this end, we choose the following expressions:

$$b := \sum \{\alpha \in \mathsf{At} \mid h(\alpha) \in \Sigma \times X\} \qquad \ell := \Big(\mathop{+}_{\alpha \leq b} \lfloor h(\alpha) \rfloor_{s^{\mathcal{Y}}} \Big)^{(b)} \cdot \mathop{+}_{\alpha \leq \overline{b}} \lfloor h(\alpha) \rfloor_{s^{\mathcal{Z}}}$$

We can then define s by setting $s(x) = s^{\mathcal{Y}}(x) \cdot \ell$ for $x \in Y$, and $s(x) = s^{\mathcal{Z}}(x)$ for $x \in Z$. A detailed argument showing that s is a solution can be found in the extended version of the paper [Smolka et al. 2019a]. □

As it turns out, we can do a round-trip, showing that the solution to the (initial state of the) Thompson automaton for an expression is equivalent to the original expression.

Theorem 4.8 (Correctness II). *Let $e \in \mathsf{Exp}$. Then $\mathcal{X}_e^{\iota}$ admits a solution s such that $e \equiv s(\iota)$.*

Finally, we show that the automata construction of the previous section gives well-nested automata.

Theorem 4.9 (well-nestedness of Thompson construction). *$\mathcal{X}_e$ and $\mathcal{X}_e^{\iota}$ are well-nested for all expressions e.*

Proof. We proceed by induction on e. In the base, let $\mathcal{Z} = \langle \varnothing, \varnothing \rangle$ and $\mathcal{I} = \langle \{*\}, * \mapsto \mathbf{1} \rangle$ denote the coalgebras with no states and with a single all-accepting state, respectively. Note that $\mathcal{Z}$ and $\mathcal{I}$ are well-nested, and that for $b \in \mathsf{BExp}$ and $p \in \Sigma$ we have $\mathcal{X}_b = \mathcal{Z}$ and $\mathcal{X}_p = \mathcal{I}$.

All of the operations used to build $\mathcal{X}_e$, as detailed in Figure 4, can be phrased in terms of an appropriate uniform continuation of a coproduct; for instance, when $e = f^{(b)}$ we have that $\mathcal{X}_e = (\mathcal{X}_f + \mathcal{I})[X_f, \iota_e]$. Consequently, the Thompson automaton $\mathcal{X}_e$ is well-nested by construction. Finally, observe that $\mathcal{X}_e^{\iota} = (\mathcal{I} + \mathcal{X}_e)[\{*\}, \iota_e]$; hence, $\mathcal{X}_e^{\iota}$ is well-nested as well. □

Theorems 4.1, 4.7 and 4.9 now give us the desired Kleene theorem.

Corollary 4.10 (Kleene Theorem). *Let $L \subseteq \mathsf{GS}$. The following are equivalent:*

(1) $L = [\![e]\!]$ for a GKAT expression e.
(2) $L = \ell^{\mathcal{X}}(\iota)$ for a well-nested, finite-state G-automaton $\mathcal{X}$ with initial state ι.

5 DECISION PROCEDURE

We saw in the last section that GKAT expressions can be efficiently converted to equivalent automata with a linear number of states. Equivalence of automata can be established algorithmically, supporting a decision procedure for GKAT that is significantly more efficient than decision procedures for KAT. In this section, we describe our algorithm.

First, we define bisimilarity of automata states in the usual way [Kozen and Tseng 2008].

Definition 5.1 (Bisimilarity). Let $\mathcal{X}$ and $\mathcal{Y}$ be G-coalgebras. A *bisimulation* between $\mathcal{X}$ and $\mathcal{Y}$ is a binary relation $R \subseteq X \times Y$ such that if $x\ R\ y$, then the following implications hold:

(i) if $\delta^{\mathcal{X}}(x)(\alpha) \in 2$, then $\delta^{\mathcal{Y}}(y)(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$; and
(ii) if $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$, then $\delta^{\mathcal{Y}}(y)(\alpha) = (p, y')$ and $x'\ R\ y'$ for some y'.

States x and y are called *bisimilar*, denoted $x \sim y$, if there exists a bisimulation relating x and y.

As usual, we would like to reduce automata equivalence to bisimilarity. It is easy to see that bisimilar states recognize the same language.

Lemma 5.2. *If $\mathcal{X}$ and $\mathcal{Y}$ are G-coalgebras with bisimilar states $x \sim y$, then $\ell^{\mathcal{X}}(x) = \ell^{\mathcal{Y}}(y)$.*

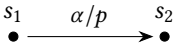
Proof. We verify that $w \in \ell^{\mathcal{X}}(x) \Leftrightarrow w \in \ell^{\mathcal{Y}}(y)$ by induction on the length of $w \in \mathrm{GS}$:

- For $\alpha \in \mathrm{GS}$, we have $\alpha \in \ell^{\mathcal{X}}(x) \Leftrightarrow \delta^{\mathcal{X}}(x)(\alpha) = 1 \Leftrightarrow \delta^{\mathcal{Y}}(y)(\alpha) = 1 \Leftrightarrow \alpha \in \ell^{\mathcal{Y}}(y)$.
- For $\alpha p w \in \mathrm{GS}$, we use bisimilarity and the induction hypothesis to derive

$$\begin{aligned} \alpha p w \in \ell^{\mathcal{X}}(x) &\iff \exists x'.\ \delta^{\mathcal{X}}(x)(\alpha) = (p, x') \wedge w \in \ell^{\mathcal{X}}(x') \\ &\iff \exists y'.\ \delta^{\mathcal{Y}}(y)(\alpha) = (p, y') \wedge w \in \ell^{\mathcal{Y}}(y') \iff \alpha p w \in \ell^{\mathcal{Y}}(y). \end{aligned}$$

□

The converse direction, however, does not hold for G-coalgebras in general. To see the problem, consider the following automaton, where $\alpha \in \mathrm{At}$ is an atom and $p \in \Sigma$ is an action:



Both states recognize the empty language, that is *i.e.*, $\ell(s_1) = \ell(s_2) = \varnothing$; but s_2 rejects immediately, whereas s_1 may first take a transition. As a result, s_1 and s_2 are not bisimilar. Intuitively, the language accepted by a state does not distinguish between early and late rejection, whereas bisimilarity does. We solve this by disallowing late rejection, *i.e.*, transitions that can never lead to an accepting state; we call coalgebras that respect this restriction *normal*.

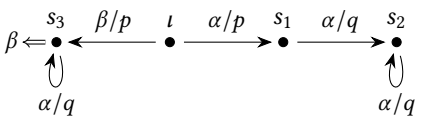
5.1 Normal Coalgebras

We classify states and coalgebras as follows.

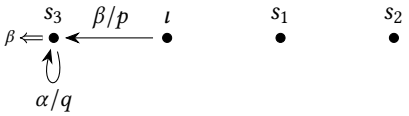
Definition 5.3 (Live, Dead, Normal). Let $\mathcal{X} = \langle X, \delta^{\mathcal{X}} \rangle$ denote a G-coalgebra. A state $s \in X$ is *accepting* if $\delta^{\mathcal{X}}(s)(\alpha) = 1$ for some $\alpha \in \mathrm{At}$. A state is *live* if it can transition to an accepting state one or more steps, or *dead* otherwise. A coalgebra is *normal* if it has no transitions to dead states.

Remark 5.4. Note that, equivalently, a state is live iff $\ell^{\mathcal{X}}(s) \neq \varnothing$ and dead iff $\ell^{\mathcal{X}}(s) = \varnothing$. Dead states can exist in a normal coalgebra, but they must immediately reject all $\alpha \in \mathrm{At}$, since any successor of a dead state would also be dead.

Example 5.5. Consider the following automaton.



The state s_3 is accepting. The states ι and s_3 are live, since they can reach an accepting state. The states s_1 and s_2 are dead, since they can only reach non-accepting states. The automaton is not normal, since it contains the transitions $\iota \xrightarrow{\alpha/p} s_1$, $s_1 \xrightarrow{\alpha/q} s_2$, and $s_2 \xrightarrow{\alpha/q} s_2$ to dead states s_1 and s_2. We can *normalize* the automaton by removing these transitions:



The resulting automaton is normal: the dead states s_1 and s_2 reject all $\alpha \in \mathrm{At}$ immediately. □

The example shows how G-coalgebra can be normalized. Formally, let $\mathcal{X} = \langle X, \delta \rangle$ denote a coalgebra with dead states $D \subseteq X$. We define the normalized coalgebra $\widehat{\mathcal{X}} := \langle X, \widehat{\delta} \rangle$ as follows:

$$\widehat{\delta}(s)(\alpha) := \begin{cases} 0 & \text{if } \delta(s)(\alpha) \in \Sigma \times D \\ \delta(s)(\alpha) & \text{otherwise.} \end{cases}$$

Lemma 5.6 (Correctness of normalization). *Let $\mathcal{X}$ be a G-coalgebra. Then the following holds:*

(i) $\mathcal{X}$ and $\widehat{\mathcal{X}}$ have the same solutions: that is, $s : X \to \mathrm{Exp}$ solves $\mathcal{X}$ if and only if s solves $\widehat{\mathcal{X}}$; and
(ii) $\mathcal{X}$ and $\widehat{\mathcal{X}}$ accept the same languages: that is, $\ell^{\mathcal{X}} = \ell^{\widehat{\mathcal{X}}}$; and
(iii) $\widehat{\mathcal{X}}$ is normal.

Proof. For the first claim, suppose s solves $\mathcal{X}$. It suffices (see the extended version [Smolka et al. 2019a]) to show that for $x \in X$ and $\alpha \in \mathrm{At}$ we have $\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$. We have two cases.

- If $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$ with x' dead, then by Proposition 4.5 we know that $[\![s(x')]\!] = \ell^{\mathcal{X}}(x') = \varnothing$. By Corollary 3.13, it follows that $s(x') \equiv 0$. Recalling that $\delta^{\widehat{\mathcal{X}}}(x)(\alpha) = 0$ by construction,

$$\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\mathcal{X}}(x)(\alpha) \rfloor_s \equiv \alpha \cdot p \cdot s(x') \equiv \alpha \cdot 0 \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$$

- Otherwise, we know that $\delta^{\widehat{\mathcal{X}}}(x)(\alpha) = \delta^{\mathcal{X}}(x)(\alpha)$, and thus

$$\alpha \cdot s(x) \equiv \alpha \cdot \lfloor \delta^{\mathcal{X}}(x)(\alpha) \rfloor_s \equiv \alpha \cdot \lfloor \delta^{\widehat{\mathcal{X}}}(x)(\alpha) \rfloor_s$$

The other direction of the first claim can be shown analogously.

For the second claim, we can establish $x \in \ell^{\mathcal{X}}(s) \Leftrightarrow x \in \ell^{\widehat{\mathcal{X}}}(s)$ for all states s by a straightforward induction on the length of $x \in \mathrm{GS}$, using that dead states accept the empty language.

For the third claim, we note that the dead states of $\mathcal{X}$ and $\widehat{\mathcal{X}}$ coincide by claim two; thus $\widehat{\mathcal{X}}$ has no transition to dead states by construction. □

5.2 Bisimilarity for Normal Coalgebras

We would like to show that, for normal coalgebras, states are bisimilar if and only if they accept the same language. This will allow us to reduce language-equivalence to bisimilarity, which is easy to establish algorithmically. We need to take a slight detour.

Recall the determinacy property satisfied by GKAT languages (Definition 2.2): a language $L \subseteq \mathrm{GS}$ is deterministic if, whenever strings $x, y \in L$ agree on the first n atoms, they also agree on the first n actions (or absence thereof). Now, let $\mathcal{L} \subseteq 2^{\mathrm{GS}}$ denote the set of deterministic languages. $\mathcal{L}$ carries

a coalgebra structure $\langle \mathcal{L}, \delta^{\mathcal{L}} \rangle$ whose transition map $\delta^{\mathcal{L}}$ is the semantic Brzozowski derivative:

$$\delta^{\mathcal{L}}(L)(\alpha) := \begin{cases} (p, \{x \in \mathrm{GS} \mid \alpha p x \in L\}) & \text{if } \{x \in \mathrm{GS} \mid \alpha p x \in L\} \neq \varnothing \\ 1 & \text{if } \alpha \in L \\ 0 & \text{otherwise.} \end{cases}$$

Note that the map is well-defined by determinacy: precisely one of the three cases holds.

Next, we define *structure-preserving maps* between G-coalgebras in the usual way:

Definition 5.7 (Homomorphism). A homomorphism between G-coalgebras $\mathcal{X}$ and $\mathcal{Y}$ is a map $h\colon X \to Y$ from states of $\mathcal{X}$ to states of $\mathcal{Y}$ that respects the transition structures in the following sense:

$$\delta^{\mathcal{Y}}(h(x)) = (Gh)(\delta^{\mathcal{X}}(x)).$$

More concretely, for all $\alpha \in \mathrm{At}$, $p \in \Sigma$, and $x, x' \in X$,

(i) if $\delta^{\mathcal{X}}(x)(\alpha) \in 2$, then $\delta^{\mathcal{Y}}(h(x))(\alpha) = \delta^{X}(x)(\alpha)$; and
(ii) if $\delta^{\mathcal{X}}(x)(\alpha) = (p, x')$, then $\delta^{\mathcal{Y}}(h(x))(\alpha) = (p, h(x'))$. □

We can now show that the acceptance map $\ell^{\mathcal{X}}\colon X \to 2^{\mathrm{GS}}$ is structure-preserving in the above sense. Moreover, it is the *only* structure-preserving map from states to deterministic languages:

Theorem 5.8. *If $\mathcal{X}$ is normal, then $\ell^{\mathcal{X}}\colon X \to 2^{\mathrm{GS}}$ is the unique homomorphism $\mathcal{X} \to \mathcal{L}$.*

Since the identity function is trivially a homomorphism, Theorem 5.8 implies that $\ell^{\mathcal{L}}$ is the identity. That is, in the G-coalgebra $\mathcal{L}$, the state $L \in \mathcal{L}$ accepts the language L! This proves that every deterministic language is recognized by a G-coalgebra, possibly with an infinite number of states.

Theorem 5.8 says that $\mathcal{L}$ is *final* for normal G-coalgebras. The desired connection between bisimilarity and language-equivalence is then a standard corollary [Rutten 2000]:

Corollary 5.9. *Let $\mathcal{X}$ and $\mathcal{Y}$ be normal with states s and t. Then $s \sim t$ if and only if $\ell^{\mathcal{X}}(s) = \ell^{\mathcal{Y}}(t)$.*

Proof. The implication from left to right is Lemma 5.2. For the other implication, we observe that the relation $R := \{(s, t) \in X \times Y \mid \ell^{\mathcal{X}}(s) = \ell^{\mathcal{Y}}(t)\}$ is a bisimulation, using that $\ell^{\mathcal{X}}$ and $\ell^{\mathcal{Y}}$ are homomorphisms by Theorem 5.8:

- Suppose $s\ R\ t$ and $\delta^{\mathcal{X}}(s)(\alpha) \in 2$. Then $\delta^{\mathcal{X}}(s)(\alpha) = \delta^{\mathcal{L}}(\ell^{\mathcal{X}}(s))(\alpha) = \delta^{\mathcal{L}}(\ell^{\mathcal{Y}}(t))(\alpha) = \delta^{\mathcal{Y}}(t)(\alpha)$.
- Suppose $s\ R\ t$ and $\delta^{\mathcal{X}}(s)(\alpha) = (p, s')$. Then $\delta^{\mathcal{L}}(\ell^{\mathcal{Y}}(t))(\alpha) = \delta^{\mathcal{L}}(\ell^{\mathcal{X}}(s))(\alpha) = (p, \ell^{\mathcal{X}}(s'))$. This implies that $\delta^{\mathcal{Y}}(t)(\alpha) = (p, t')$ for some t', using that $\ell^{\mathcal{Y}}$ is a homomorphism. Hence

$$(p, \ell^{\mathcal{Y}}(t')) = \delta^{\mathcal{L}}(\ell^{\mathcal{Y}}(t))(\alpha) = (p, \ell^{\mathcal{X}}(s'))$$

by the above equation, which implies $s'\ R\ t'$ as required. □

We prove a stronger version of this result in the extended version of this paper [Smolka et al. 2019a].

5.3 Deciding Equivalence

We now have all the ingredients required for deciding efficiently whether two expressions are equivalent. Given two expressions e_1 and e_2, the algorithm proceeds as follows:

(1) Convert e_1 and e_2 to equivalent Thompson automata $\mathcal{X}_1$ and $\mathcal{X}_2$;
(2) Normalize the automata, obtaining $\widehat{\mathcal{X}_1}$ and $\widehat{\mathcal{X}_2}$;
(3) Check bisimilarity of the start states ι_1 and ι_2 using Hopcroft-Karp (see Algorithm 1);
(4) Return **true** if $\iota_1 \sim \iota_2$, otherwise return **false**.

Theorem 5.10. *The above algorithm decides whether $[\![e_1]\!] = [\![e_2]\!]$ in time $O(n \cdot \alpha(n))$ for $|\mathrm{At}|$ constant, where α denotes the inverse Ackermann function and $n = |e_1| + |e_2|$.*

Algorithm 1: Hopcroft and Karp's algorithm [Hopcroft and Karp 1971], adapted to G-automata.

Input: G-automata $\mathcal{X} = \langle X, \delta^{\mathcal{X}}, \iota^{\mathcal{X}} \rangle$ and $\mathcal{Y} = \langle Y, \delta^{\mathcal{Y}}, \iota^{\mathcal{Y}} \rangle$, finite and normal; X, Y disjoint.
Output: **true** if $\mathcal{X}$ and $\mathcal{Y}$ are equivalent, **false** otherwise.

```
1  todo ← Queue.singleton(ι^X, ι^Y);            // state pairs that need to be checked
2  forest ← UnionFind.disjointForest(X ⊎ Y);
3  while not todo.isEmpty() do
4      x, y ← todo.pop();
5      r_x, r_y ← forest.find(x), forest.find(y);
6      if r_x = r_y then continue;                  // safe to assume bisimilar
7      for α ∈ At do                                   // check Definition 5.1
8          switch δ^X(x)(α), δ^Y(y)(α) do
9              case b_1, b_2 with b_1 = b_2 do       // case (i) of Definition 5.1
10                 continue
11             case (p, x'), (p, y') do             // case (ii) of Definition 5.1
12                 todo.push(x', y')
13             otherwise do return false;                    // not bisimilar
14         end
15     end
16     forest.union(r_x, r_y);                          // mark as bisimilar
17 end
18 return true;
```

Proof. The algorithm is correct by Theorem 4.1, Lemma 5.6, and Corollary 5.9:

$$[\![e_1]\!] = [\![e_2]\!] \iff \ell^{\mathcal{X}_1}(\iota_1) = \ell^{\mathcal{X}_2}(\iota_2) \iff \ell^{\widehat{\mathcal{X}}_1}(\iota_1) = \ell^{\widehat{\mathcal{X}}_2}(\iota_2) \iff \iota_1 \sim \iota_2$$

For the complexity claim, we analyze the running time of steps (1)–(3) of the algorithm:

(1) Recall by Proposition 4.2 that the Thompson construction converts e_i to an automaton with $O(|e_i|)$ states in time $O(|e_i|)$. Hence this step takes time $O(n)$.
(2) Normalizing $\mathcal{X}_i$ amounts to computing its dead states. This requires time $O(|e_i|)$ using a breadth-first traversal as follows (since there are at most $|\mathrm{At}| \in O(1)$ transitions per state). We find all states that can reach an accepting state by first marking all accepting states, and then performing a reverse breadth-first search rooted at the accepting states. All marked states are then live; all unmarked states are dead.
(3) Since $\widehat{\mathcal{X}}_i$ has $O(|e_i|)$ states and there are at most $|\mathrm{At}| \in O(1)$ transitions per state, Hopcroft-Karp requires time $O(n \cdot \alpha(n))$ by a classic result due to Tarjan [1975]. □

Theorem 5.10 establishes a stark complexity gap with KAT, where the same decision problem is PSPACE-complete [Cohen et al. 1996] even for a constant number of atoms. Intuitively, the gap arises because GKAT expressions can be translated to linear-size deterministic automata, whereas KAT expressions may require exponential-size deterministic automata.

A shortcoming of Algorithm 1 is that it may scale poorly if the number of atoms $|\mathrm{At}|$ is large. It is worth noting that there are symbolic variants [Pous 2015] of the algorithm that avoid enumerating At explicitly (*cf.* Line 7 of Algorithm 1), and can often scale to very large alphabets in practice. As a concrete example, a version of GKAT specialized to probabilistic network verification was recently

shown [Smolka et al. 2019b] to scale to data-center networks with thousands of switches. In the worst case, however, we have the following hardness result:

Proposition 5.11. *If* $|\mathsf{At}|$ *is not a constant, GKAT equivalence is co-NP-hard, but in PSPACE.*

Proof. For the hardness result, we observe that Boolean *un*satisfiability reduces to GKAT equivalence: $b \in \mathsf{BExp}$ is unsatisfiable, interpreting the primitive tests as variables, iff $[\![b]\!] = \varnothing$. The PSPACE upper bound is inherited from KAT by Remark 2.1. □

6 COMPLETENESS FOR THE LANGUAGE MODEL

In Section 3 we presented an axiomatization that is sound with respect to the language model, and put forward the conjecture that it is also complete. We have already proven a partial completeness result (Corollary 3.13). In this section, we return to this matter and show we can prove completeness with a generalized version of (W3).

6.1 Systems of Left-affine Equations

A *system of left-affine equations* (or simply, a *system*) is a finite set of equations of the form

$$\begin{aligned} \boldsymbol{x_1} &= e_{11}\boldsymbol{x_1} +_{b_{11}} \cdots +_{b_{1(n-1)}} e_{1n}\boldsymbol{x_n} +_{b_{1n}} d_1 \\ &\vdots \\ \boldsymbol{x_n} &= e_{n1}\boldsymbol{x_1} +_{b_{n1}} \cdots +_{b_{n(n-1)}} e_{nn}\boldsymbol{x_n} +_{b_{nn}} d_n \end{aligned} \tag{3}$$

where $+_b$ associates to the right, the $\boldsymbol{x_i}$ are variables, the e_{ij} are GKAT expressions, and the b_{ij} and d_i are Boolean guards satisfying the following row-wise disjointness property for row $1 \le i \le n$:

- for all $j \neq k$, the guards b_{ij} and b_{ik} are disjoint: $b_{ij} \cdot b_{ik} \equiv_{\mathrm{BA}} 0$; and
- for all $1 \le j \le n$, the guards b_{ij} and d_i are disjoint: $b_{ij} \cdot d_i \equiv_{\mathrm{BA}} 0$.

Note that by the disjointness property, the ordering of the summands is irrelevant: the system is invariant (up to $\equiv$) under column permutations. A *solution* to such a system is a function $s : \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\} \to \mathsf{Exp}$ assigning expressions to variables such that, for row $1 \le i \le n$:

$$s(\boldsymbol{x_i}) \equiv e_{i1} \cdot s(\boldsymbol{x_1}) +_{b_{i1}} \cdots +_{b_{i(n-1)}} e_{in} \cdot s(\boldsymbol{x_n}) +_{b_{in}} d_i$$

Note that any finite G-coalgebra gives rise to a system where each variable represents a state, and the equations define what it means to be a solution to the coalgebra (c.f. Definition 4.3); indeed, a solution to a G-coalgebra is precisely a solution to its corresponding system of equations, and vice versa. In particular, for a coalgebra $\mathcal{X}$ with states x_1 to x_n, the parameters for equation i are:

$$b_{ij} = \sum \{\alpha \in \mathsf{At} \mid \delta^{\mathcal{X}}(x_i)(\alpha) \in \Sigma \times \{x_j\}\}$$

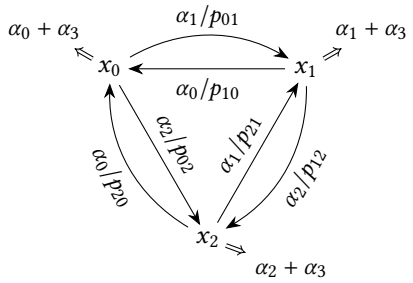
$$d_i = \sum \{\alpha \in \mathsf{At} \mid \delta^{\mathcal{X}}(x_i)(\alpha) = 1\} \qquad e_{ij} = \mathop{+}_{\alpha\,:\,\delta^{\mathcal{X}}(x_i)(\alpha)=(p_\alpha, x_j)} p_\alpha$$

Systems arising from G-coalgebras have a useful property: for all e_{ij}, it holds that $E(e_{ij}) \equiv 0$. This property is analogous to the *empty word property* in Salomaa's axiomatization of regular languages [Salomaa 1966]; we call such systems *Salomaa*.

To obtain a general completeness result beyond Section 3.4, we assume an additional axiom:

> *Uniqueness axiom.* Any system of left-affine equations that is Salomaa has *at most* one solution, modulo $\equiv$. More precisely, whenever s and s' are solutions to a Salomaa system, it holds that $s(x_i) \equiv s'(x_i)$ for all $1 \le i \le n$.

Remark 6.1. We do not assume that a solution always exists, but only that if it does, then it is unique up to ≡. It would be unsound to assume that all such systems have solutions; the following automaton and its system, due to [Kozen and Tseng 2008], constitutes a counterexample:



$$\begin{aligned}\boldsymbol{x_0} &\equiv p_{01}\boldsymbol{x_1} +_{\alpha_1} p_{02}\boldsymbol{x_2} +_{\alpha_2} (\alpha_0 + \alpha_3)\\ \boldsymbol{x_1} &\equiv p_{10}\boldsymbol{x_0} +_{\alpha_0} p_{12}\boldsymbol{x_2} +_{\alpha_2} (\alpha_1 + \alpha_3)\\ \boldsymbol{x_2} &\equiv p_{20}\boldsymbol{x_0} +_{\alpha_1} p_{21}\boldsymbol{x_1} +_{\alpha_0} (\alpha_2 + \alpha_3)\end{aligned}$$

As shown in [Kozen and Tseng 2008], no corresponding while program exists for this system.

When $n = 1$, a system is a single equation of the form $x = ex +_b d$. In this case, (W1) tells us that a solution does exist, namely $e^{(b)}d$, and (W3) says that this solution is unique up to ≡ under the proviso $E(e) \equiv 0$. In this sense, we can regard the uniqueness axiom as a generalization of (W3) to systems with multiple variables.

Theorem 6.2. *The uniqueness axiom is sound in the model of guarded strings: given a system of left-affine equations as in* (3) *that is Salomaa, there exists at most one* $R : \{x_1, \ldots, x_n\} \to 2^{\mathrm{GS}}$ *s.t.*

$$R(x_i) = \left(\bigcup_{1 \le j \le n} [\![b_{ij}]\!] \diamond [\![e_{ij}]\!] \diamond R(x_j) \right) \cup [\![d_i]\!]$$

Proof Sketch. We recast this system as a matrix-vector equation of the form $x = Mx + D$ in the KAT of n-by-n matrices over 2^{GS}; solutions to x in this equation are in one-to-one correspondence with functions R as above. We then show that the map $\sigma(x) = Mx + D$ on the set of n-dimensional vectors over 2^{GS} is contractive in a certain metric, and therefore has a unique fixpoint by the Banach fixpoint theorem; hence, there can be at most one solution x. □

6.2 General Completeness

Using the generalized version of the fixpoint axiom, we can now establish completeness.

Theorem 6.3 (Completeness). *The axioms are complete w.r.t.* $[\![-]\!]$*: given* $e_1, e_2 \in \mathsf{Exp}$,

$$[\![e_1]\!] = [\![e_2]\!] \implies e_1 \equiv e_2.$$

Proof. Let $\mathcal{X}_1$ and X_2 be the Thompson automata corresponding to e_1 and e_2, with initial states ι_1 and ι_2, respectively. Theorem 4.8 shows there are solutions s_1 and s_2, with $s_1(\iota_1) \equiv e_1$ and $s_2(\iota_2) \equiv e_2$; and we know from Lemma 5.6 that s_1 and s_2 solve the normalized automata $\widehat{\mathcal{X}_1}$ and $\widehat{\mathcal{X}_2}$. By Lemma 5.6, Theorem 4.1, and the premise, we derive that $\widehat{\mathcal{X}_1}$ and $\widehat{\mathcal{X}_2}$ accept the same language:

$$\ell^{\widehat{\mathcal{X}_1}}(\iota_1) = \ell^{\mathcal{X}_1}(\iota_1) = [\![e_1]\!] = [\![e_2]\!] = \ell^{\mathcal{X}_2}(\iota_2) = \ell^{\widehat{\mathcal{X}_2}}(\iota_2).$$

This implies, by Corollary 5.9, that there is a bisimulation R between $\widehat{\mathcal{X}_1}$ and $\widehat{\mathcal{X}_2}$ relating ι_1 and ι_2. This bisimulation can be given the following transition structure,

$$\delta^{\mathcal{R}}(x_1, x_2)(\alpha) := \begin{cases} 0 & \text{if } \delta^{\widehat{\mathcal{X}_1}}(x_1)(\alpha) = 0 \text{ and } \delta^{\widehat{\mathcal{X}_2}}(x_2)(\alpha) = 0 \\ 1 & \text{if } \delta^{\widehat{\mathcal{X}_1}}(x_1)(\alpha) = 1 \text{ and } \delta^{\widehat{\mathcal{X}_2}}(x_2)(\alpha) = 1 \\ (p, (x_1', x_2')) & \text{if } \delta^{\widehat{\mathcal{X}_1}}(x_1)(\alpha) = (p, x_1') \text{ and } \delta^{\widehat{\mathcal{X}_2}}(x_2)(\alpha) = (p, x_2') \end{cases}$$

turning $\mathcal{R} = \langle R, \delta^{\mathcal{R}} \rangle$ into a G-coalgebra; note that $\delta^{\mathcal{R}}$ is well-defined since R is a bisimulation.

Now, define $s_1', s_2' : R \to \mathsf{Exp}$ by $s_1'(x_1, x_2) = s_1(x_1)$ and $s_2'(x_1, x_2) = s_2(x_2)$. We claim that s_1' and s_2' are both solutions to $\mathcal{R}$; to see this, note that for $\alpha \in \mathsf{At}$, $(x_1, x_2) \in R$, and $i \in \{1, 2\}$, we have that

$$\begin{aligned}
\alpha \cdot s_i'(x_i, x_i) &\equiv \alpha \cdot s_i(x_i) && \text{(Def. } s_i'\text{)} \\
&\equiv \alpha \cdot \lfloor \delta^{\widehat{X_i}}(x_i)(\alpha) \rfloor_{s_i} && (s_i \text{ solves } \widehat{X_i}) \\
&\equiv \alpha \cdot \lfloor \delta^{\mathcal{R}}(x_1, x_2)(\alpha) \rfloor_{s_i'} && \text{(Def. } s_i' \text{ and } \lfloor - \rfloor\text{)}
\end{aligned}$$

Thus, s_i' is a solution (c.f. the extended proof of Lemma 5.6 [Smolka et al. 2019a]).

Since the system of left-affine equations induced by $\mathcal{R}$ is Salomaa, the uniqueness axiom then tells us that $s_1(\iota_1) = s_1'(\iota_1, \iota_2) \equiv s_2'(\iota_1, \iota_2) = s_2(\iota_2)$; hence, we can conclude that $e_1 \equiv e_2$. □

7 COALGEBRAIC STRUCTURE

The coalgebraic theory of GKAT is quite different from that of KA and KAT because the final G-coalgebra, without the normality assumption from § 5.1, is not characterized by sets of finite guarded strings. Even including infinite accepted strings is not enough, as this still cannot distinguish between early and late rejection. It is therefore of interest to characterize the final G-coalgebra and determine its precise relationship to the language model. We give a brief overview of these results, which give insight into the nature of halting versus looping and underscore the role of topology in coequational specifications.

In the extended version of the paper [Smolka et al. 2019a] we give two characterizations of the final G-coalgebra, one in terms of nonexpansive maps $\mathsf{At}^\omega \to \Sigma^* \cup \Sigma^\omega$ with natural metrics defined on both spaces and one in terms of labeled trees with nodes indexed by At^+, and show their equivalence. We also state and prove a stronger form of the bisimilarity lemma (Corollary 5.9).

We have discussed the importance of the determinacy property (Definition 2.2). In the extended version [Smolka et al. 2019a] we identify another important property satisfied by all languages $L^X(s)$, a certain closure property defined in terms of a natural topology on At^ω. We define a language model $\mathcal{L}'$, a G-coalgebra whose states are the subsets of $\mathsf{GS} \cup \omega\text{-GS}$ satisfying the determinacy and closure properties and whose transition structure is the semantic Brzozowski derivative:

$$\delta^{\mathcal{L}'}(A)(\alpha) = \begin{cases} (p, \{x \mid \alpha p x \in A\}) & \text{if } \{x \mid \alpha p x \in A\} \neq \varnothing \\ 1 & \text{if } \alpha \in A \\ 0 & \text{otherwise.} \end{cases}$$

Although this looks similar to the language model $\mathcal{L}$ of Section 5.2, they are not the same: states of $\mathcal{L}$ contain finite strings only, and $\mathcal{L}$ and $\mathcal{L}'$ are not isomorphic.

We show that L is identity on $\mathcal{L}'$ and that $\mathcal{L}'$ is isomorphic to a subcoalgebra of the final G-coalgebra. It is not the final G-coalgebra, because early and late rejection are not distinguished: an automaton could transition before rejecting or reject immediately. Hence, $L : (X, \delta^X) \to \mathcal{L}'$ is not a homomorphism in general. However, normality prevents this behavior, and L is a homomorphism if (X, δ^X) is normal. Thus $\mathcal{L}'$ contains the unique homomorphic image of all normal G-coalgebras.

Finally, we identify a subcoalgebra $\mathcal{L}'' \subseteq \mathcal{L}'$ that is normal and final in the category of normal G-coalgebras. The subcoalgebra $\mathcal{L}''$ is defined topologically; roughly speaking, it consists of sets $A \subseteq \mathsf{GS} \cup \omega\text{-GS}$ such that A is the topological closure of $A \cap \mathsf{GS}$. Thus $\mathcal{L}''$ is isomorphic to the language model $\mathcal{L}$ of Section 5.2: the states of $\mathcal{L}$ are obtained from those of $\mathcal{L}''$ by intersecting with GS, and the states of $\mathcal{L}''$ are obtained from those of $\mathcal{L}$ by taking the topological closure. Thus $\mathcal{L}$ is isomorphic to a coequationally-defined subcoalgebra of the final G-coalgebra.

We also remark that $\mathcal{L}'$ itself is final in the category of G-coalgebras that satisfy a weaker property than normality, the so-called *early failure property*, which can also be characterized topologically.

8 RELATED WORK

Program schematology is one of the oldest areas of study in the mathematics of computing. It is concerned with questions of translation and representability among and within classes of program schemes, such as flowcharts, while programs, recursion schemes, and schemes with various data structures such as counters, stacks, queues, and dictionaries [Garland and Luckham 1973; Ianov 1960; Luckham et al. 1970; Paterson and Hewitt 1970; Rutledge 1964; Shepherdson and Sturgis 1963]. A classical pursuit in this area was to find mechanisms to transform unstructured flowcharts to structured form [Ashcroft and Manna 1972; Böhm and Jacopini 1966; Kosaraju 1973; Morris et al. 1997; Oulsnam 1982; Peterson et al. 1973; Ramshaw 1988; Williams and Ossher 1978]. A seminal result was the *Böhm-Jacopini theorem* [Böhm and Jacopini 1966], which established that all flowcharts can be converted to while programs provided auxiliary variables are introduced. Böhm and Jacopini conjectured that the use of auxiliary variables was necessary in general, and this conjecture was confirmed independently by Ashcroft and Manna [1972] and Kosaraju [1973].

Early results in program schematology, including those of [Ashcroft and Manna 1972; Böhm and Jacopini 1966; Kosaraju 1973], were typically formulated at the first-order uninterpreted (schematic) level. However, many restructuring operations can be accomplished without reference to first-order constructs. It was shown in [Kozen and Tseng 2008] that a purely propositional formulation of the Böhm-Jacopini theorem is false: there is a three-state deterministic propositional flowchart that is not equivalent to any propositional while program. As observed by a number of authors (*e.g.* [Kosaraju 1973; Peterson et al. 1973]), while loops with multi-level breaks are sufficient to represent all deterministic flowcharts without introducing auxiliary variables, and [Kosaraju 1973] established a strict hierarchy based on the allowed levels of the multi-level breaks. That result was reformulated and proved at the propositional level in [Kozen and Tseng 2008].

The notions of functions on a domain, variables ranging over that domain, and variable assignment are inherent in first-order logic, but are absent at the propositional level. Moreover, many arguments rely on combinatorial graph restructuring operations, which are difficult to formalize. Thus the value of the propositional view is twofold: it operates at a higher level of abstraction and brings topological and coalgebraic concepts and techniques to bear.

Propositional while programs and their encoding in terms of the regular operators goes back to early work on Propositional Dynamic Logic [Fischer and Ladner 1979]. GKAT as an independent system and its semantics were introduced in [Kozen 2008; Kozen and Tseng 2008] under the name *propositional while programs*, although the succinct form of the program operators is new here. Also introduced in [Kozen 2008; Kozen and Tseng 2008] were the functor G and automaton model (Section 4), the determinacy property (Definition 2.2) (called *strict determinacy* there), and the concept of normality (Section 5.1) (called *liveness* there). The linear construction of an automaton from a while program was sketched in [Kozen 2008; Kozen and Tseng 2008], based on earlier results for KAT automata [Kozen 2003], but the complexity of deciding equivalence was not addressed. The more rigorous alternative construction given here (Section 4.2) is needed to establish well-nestedness, thereby enabling our Kleene theorem. The existence of a complete axiomatization was not considered in [Kozen 2008; Kozen and Tseng 2008].

Guarded strings, which form the basis of our language semantics, come from [Kaplan 1969].

The axiomatization we propose for GKAT is closely related to Salomaa's axiomatization of regular expressions based on unique fixed points [Salomaa 1966] and to Silva's coalgebraic generalization of KA [Silva 2010]. The proof technique we used for completeness is inspired by [Silva 2010].

The relational semantics is inspired by that for KAT [Kozen and Smith 1996], which goes back to work on Dynamic Logic [Fischer and Ladner 1979]. Because the fixpoint axiom uses a non-algebraic side condition, extra care is needed to define the relational interpretation for GKAT.

9 CONCLUSIONS AND FUTURE DIRECTIONS

We have presented a comprehensive algebraic and coalgebraic study of GKAT, an abstract programming language with uninterpreted actions. Our main contributions include: (i) a new automata construction yielding a nearly linear time decidability result for program equivalence; (ii) a Kleene theorem for GKAT providing an exact correspondence between programs and a well-defined class of automata; and (iii) a set of sound and complete axioms for program equivalence.

We hope this paper is only the beginning of a long and beautiful journey into understanding the (co)algebraic properties of efficient fragments of imperative programming languages. We briefly discuss some limitations of our current development and our vision for future work.

As in Salomaa's axiomatization of KA, our axiomatization is not fully algebraic: the side condition of (W3) is only sensible for the language model. As a result, the current completeness proof does not generalize to other natural models of interest—*e.g.*, probabilistic or relational. To overcome this limitation, we would like to adapt Kozen's axiomatization of KA to GKAT by developing a natural order for GKAT programs. In the case of KA we have $e \leq f \ :\Longleftrightarrow\ e + f = f$, but this natural order is no longer definable in the absence of + and so we need to axiomatize $e \leq f$ for GKAT programs directly. This appears to be the main missing piece to obtain an algebraic axiomatization.

On the coalgebraic side, we are interested in studying the different classes of G-coalgebras from a coequational perspective. Normal coalgebras, for instance, form a covariety, and hence are characterized by coequations. If well-nested G-coalgebras could be shown to form a covariety, this would imply completeness of the axioms without the extra uniqueness axiom from Section 6.

Various extensions of KAT to reason about richer programs (KAT+B!, NetKAT, ProbNetKAT) have been proposed, and it is natural to ask whether extending GKAT in similar directions will yield interesting algebraic theories and decision procedures for domain-specific applications. For instance, recent work [Smolka et al. 2019b] on a probabilistic network verification tool suggests that GKAT is better suited for probabilistic models than KAT, as it avoids mixing non-determinism and probabilities. The complex semantics of probabilistic programs would make a framework for equational and automated reasoning especially valuable.

In a different direction, a language model containing infinite traces could be interesting in many applications, as it could serve as a model to reason about non-terminating programs—*e.g.*, loops in NetKAT in which packets may be forwarded forever. An interesting open question is whether the infinite language model can be finitely axiomatized.

Finally, another direction would be to extend the GKAT decision procedure to handle extra equations. For instance, both KAT+B! and NetKAT have independently-developed decision procedures, that are similar in flavor. This raises the question of whether the GKAT decision procedure could be extended in a more generic way, similar to the Nelson-Oppen approach [Nelson and Oppen 1979] for combining decision procedures used in SMT solving.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their feedback and help in improving this paper, and thank Paul Brunet, Fredrik Dahlqvist, and Jonathan DiLorenzo for numerous discussions and suggestions on GKAT. Jonathan DiLorenzo and Simon Docherty provided helpful feedback on drafts of this paper. We thank the Bellairs Research Institute of McGill University for providing a wonderful research environment. This work was supported in part by the University of Wisconsin, a Facebook TAV award, ERC starting grant Profoundnet (679127), a Royal Society Wolfson fellowship, a Leverhulme Prize (PLP-2016-129), NSF grants AitF-1637532, CNS-1413978, and SaTC-1717581, and gifts from Fujitsu and InfoSys.

REFERENCES

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

Allegra Angus and Dexter Kozen. 2001. *Kleene Algebra with Tests and Program Schematology*. Technical Report TR2001-1844. Computer Science Department, Cornell University.

Edward A. Ashcroft and Zohar Manna. 1972. The translation of GOTO programs into WHILE programs. In *Proc. Information Processing (IFIP)*, Vol. 1. North-Holland, Amsterdam, The Netherlands, 250–255.

Roland Backhouse. 1975. *Closure algorithms and the star-height problem of regular languages*. Ph.D. Dissertation. University of London. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.448525

Adam Barth and Dexter Kozen. 2002. *Equational Verification of Cache Blocking in LU Decomposition using Kleene Algebra with Tests*. Technical Report TR2002-1865. Computer Science Department, Cornell University.

Garrett Birkhoff and Thomas C. Bartee. 1970. *Modern applied algebra*. McGraw-Hill, New York, NY, USA.

Corrado Böhm and Guiseppe Jacopini. 1966. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Commun. ACM* (May 1966), 366–371. https://doi.org/10.1145/355592.365646

Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/2429069.2429124

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

Ernie Cohen. 1994b. *Using Kleene algebra to reason about concurrency control*. Technical Report. Telcordia, Morristown, NJ.

Ernie Cohen, Dexter Kozen, and Frederick Smith. 1996. *The complexity of Kleene algebra with tests*. Technical Report TR96-1598. Computer Science Department, Cornell University.

John Horton Conway. 1971. *Regular Algebra and Finite Machines*. Chapman and Hall, London, United Kingdom.

Ana M. Erosa and Laurie J. Hendren. 1994. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proc. Computer Languages (ICCL)*. IEEE Computer Society, Los Alamitos, CA, USA, 229–240. https://doi.org/10.1109/ICCL.1994.288377

Michael J. Fischer and Richard E. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comput. System Sci.* 18, 2 (1979), 194–211. https://doi.org/10.1016/0022-0000(79)90046-1

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Proc. European Symposium on Programming (ESOP)*. ACM, New York, NY, USA, 282–309. https://doi.org/10.1007/978-3-662-49498-1_12

Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2676726.2677011

Stephen J. Garland and David C. Luckham. 1973. Program schemes, recursion schemes, and formal languages. *J. Comput. System Sci.* 7, 2 (1973), 119 – 160. https://doi.org/10.1016/S0022-0000(73)80040-6

Michele Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*. Springer, Berlin, Heidelberg, 68–85. https://doi.org/10.1007/BFb0092872

Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. 1992. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proc. Languages and Compilers for Parallel Computing (LCPC)*. Springer, Berlin, Heidelberg, 406–420. https://doi.org/10.1007/3-540-57502-2_61

John E. Hopcroft and Richard M. Karp. 1971. *A linear algorithm for testing equivalence of finite automata*. Technical Report TR 71-114. Cornell University.

I. Ianov. 1960. The Logical Schemes of Algorithms. *Problems of Cybernetics* (1960), 82–140.

Donald M. Kaplan. 1969. Regular Expressions and the Equivalence of Programs. *J. Comput. System Sci.* 3 (1969), 361–386. https://doi.org/10.1016/S0022-0000(69)80027-9

Stephen C. Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies* (1956), 3–41.

S. Rao Kosaraju. 1973. Analysis of structured programs. In *Proc. Theory of Computing (STOC)*. ACM, New York, NY, USA, 240–252. https://doi.org/10.1145/800125.804055

Dexter Kozen. 1996. Kleene algebra with tests and commutativity conditions. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Vol. 1055. Springer-Verlag, Passau, Germany, 14–33. https://doi.org/10.1007/3-540-61042-1_35

Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

Dexter Kozen. 2003. Automata on Guarded Strings and Applications. *Matématica Contemporânea* 24 (2003), 117–139.

Dexter Kozen. 2008. Nonlocal Flow of Control and Kleene Algebra with Tests. In *Proc. Logic in Computer Science (LICS)*. IEEE, New York, NY, USA, 105–117. https://doi.org/10.1109/LICS.2008.32

Dexter Kozen and Maria-Cristina Patron. 2000. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. Computational Logic (CL) (Lecture Notes in Artificial Intelligence)*, Vol. 1861. Springer-Verlag, London, United Kingdom, 568–582. https://doi.org/10.1007/3-540-44957-4_38

Dexter Kozen and Frederick Smith. 1996. Kleene algebra with tests: Completeness and decidability. In *Proc. Computer Science Logic (CSL) (Lecture Notes in Computer Science)*, Vol. 1258. Springer-Verlag, Utrecht, The Netherlands, 244–259. https://doi.org/10.1007/3-540-63172-0_43

Dexter Kozen and Wei-Lung (Dustin) Tseng. 2008. The Böhm-Jacopini Theorem is False, Propositionally. In *Proc. Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science)*, Vol. 5133. Springer, Berlin, Heidelberg, 177–192. https://doi.org/10.1007/978-3-540-70594-9_11

David C. Luckham, David M. R. Park, and Michael S. Paterson. 1970. On formalised computer programs. *J. Comput. System Sci.* 4, 3 (1970), 220–249. https://doi.org/10.1016/S0022-0000(70)80022-8

Michael W. Mislove. 2006. On Combining Probability and Nondeterminism. *Electronic Notes in Theoretical Computer Science* 162 (2006), 261 – 265. https://doi.org/10.1016/j.entcs.2005.12.113 Proc. Algebraic Process Calculi (APC).

Paul H. Morris, Ronald A. Gray, and Robert E. Filman. 1997. GOTO Removal Based on Regular Expressions. *Journal of Software Maintenance: Research and Practice* 9, 1 (1997), 47–66. https://doi.org/10.1002/(SICI)1096-908X(199701)9:1<47::AID-SMR142>3.0.CO;2-V

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 2 (1979), 245–257. https://doi.org/10.1145/357073.357079

G. Oulsnam. 1982. Unraveling unstructured programs. *Comput. J.* 25, 3 (1982), 379–387. https://doi.org/10.1093/comjnl/25.3.379

Michael S. Paterson and Carl E. Hewitt. 1970. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, New York, NY, USA, 119–127.

W. Wesley Peterson, Tadao Kasami, and Nobuki Tokura. 1973. On the Capabilities of while, repeat, and exit Statements. *Commun. ACM* 16, 8 (1973), 503–512. https://doi.org/10.1145/355609.362337

Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *Proc. Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2676726.2677007

Lyle Ramshaw. 1988. Eliminating goto's while preserving program structure. *J. ACM* 35, 4 (1988), 893–920. https://doi.org/10.1145/48014.48021

Joseph D. Rutledge. 1964. On Ianov's Program Schemata. *J. ACM* 11, 1 (Jan. 1964), 1–9. https://doi.org/10.1145/321203.321204

Jan J. M. M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 1 (2000), 3–80. https://doi.org/10.1016/S0304-3975(00)00056-6

Arto Salomaa. 1966. Two complete axiom systems for the algebra of regular events. *J. ACM* 13, 1 (January 1966), 158–169.

John C. Shepherdson and Howard E. Sturgis. 1963. Computability of Recursive Functions. *J. ACM* 10, 2 (1963), 217–255. https://doi.org/10.1145/321160.321170

Alexandra Silva. 2010. *Kleene Coalgebra*. Ph.D. Dissertation. Radboud University.

Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2019a. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time (Extended Version). arXiv:1907.05920

Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019b. Scalable verification of probabilistic networks. In *Proc. Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 190–203. https://doi.org/10.1145/3314221.3314639

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. https://doi.org/10.1145/321879.321884

Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Daniele Varacca and Glynn Winskel. 2006. Distributing probability over non-determinism. *Mathematical Structures in Computer Science* 16, 1 (2006), 87–113. https://doi.org/10.1017/S0960129505005074

M. Williams and H. Ossher. 1978. Conversion of unstructured flow diagrams into structured form. *Comput. J.* 21, 2 (1978), 161–167. https://doi.org/10.1093/comjnl/21.2.161