Local Variable Scoping and Kleene Algebra with Tests

Kamal Aboul-Hosn and Dexter Kozen
{kamal,kozen}@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA

Abstract. We explore the power of relational semantics and equational reasoning in the style of Kleene algebra for analyzing programs with mutable, statically scoped local variables. We provide (i) a fully compositional relational semantics for a first-order programming language with constructs for local variable declaration and destructive update; and (ii) an equational proof system based on Kleene algebra with tests for proving the equivalence of programs in this language. We show that the proof system is sound and complete relative to the underlying equational theory without local variables. We illustrate the use of the system with several examples.

Keywords: Kleene algebra, Kleene algebra with tests, program verification, variable scoping, programming with state.

1 Introduction

Reasoning about programs with mutable state is an important and difficult problem. The notions of state and destructive update are fundamentally at odds with the functional style, yet remain a popular paradigm and are found in most programming languages, even nominally functional ones.

Most approaches to modeling state involve some representation of storage locations with pointers and memory cells. An early paper of Meyer and Sieber [23] introduced a framework for analyzing ALGOL procedures. This paper gave several interesting examples illustrating the subtleties of reasoning in the presence of local state. Much later attention focused on the use of denotational semantics to model a set of storage locations [12, 24, 35, 37]. The inability to prove some simple program equivalences using traditional techniques led several researchers to take a categorical approach [27, 34, 36]. See [26] for more information regarding the history of these approaches.

More recently, researchers have investigated the use of operational semantics to reason about ML programs with references. Mason and Talcott [20–22] considered a λ-calculus extended with state operations. By defining axioms in

the form of contextual assertions, Mason and Talcott were able to handle several examples of Meyer and Sieber. Pitts and Stark [29–32] also use operational semantics.

Several other recent approaches include game semantics [3–5, 19], real-time dynamic logic [11], transformational semantics [28], and various program refinement calculi [7, 13, 25], all of which attempt to capture the idea of local state in some form.

In this paper, we explore the extent to which relational semantics and equational reasoning in the style of Kleene algebra with tests (KAT) [16] can simplify the picture. KAT has previously been shown to be a mathematically rigorous, simple, and versatile system for low-level verification tasks. It is a blend of Kleene algebra (KA) and Boolean algebra that recasts several previous approaches to program verification, such as Hoare logic of partial correctness or inductive assertions, into a simple classical equational framework [6, 17, 18]. Programs in KAT are normally interpreted as binary relations over a space of valuations of program variables.

Our goal in this paper is to extend the semantics and deductive apparatus to handle local variable declarations with static scoping. We consider first-order programs as in ordinary KAT, but in addition we include a let construct

$$\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \tag{1}$$

for declaring a local variable with bounded scope. In contrast to the usual functional interpretation, the variable x in (1) is *mutable* in that it can occur on the left-hand side of an assignment $x := e$ in p. In functional languages such as ML, one must explicitly declare x to be a reference and explicitly dereference it to obtain its value, which can occasionally be awkward.

In the presence of higher-order programs, the let construct (1) can be encoded as a λ-term $(\lambda x.p)t$, but here we take (1) as primitive. The standard flat relational semantics used in first-order Kleene algebra with tests (KAT) and Dynamic Logic (DL) involving valuations of program variables is extended to accommodate the let construct. Instead of a valuation, a state consists of a stack of such valuations. The formal semantics captures the operational intuition that local variables declared in a let statement push a new valuation with finite domain, which is then popped upon exiting the scope. As in ordinary KAT, programs are interpreted as binary relations on states.

In a companion paper [2], we presented a fully compositional relational semantics for higher-order programs with destructive updates based on KAT and showed how it could be used to avoid intricate memory modeling and the explicit use of context in program equivalence proofs. We illustrated its use on several of Meyer and Sieber's benchmark examples [23], which proved to be quite amenable to this treatment. The stack-based semantics of this paper is a special case of [2], which involved a more complicated tree-like structure called a *closure structure*. However, in that paper we did not attempt to formulate an equational axiomatization; all arguments were based on semantic reasoning.

In the first-order case, we find that the let construct interacts seamlessly with the usual regular and Boolean operators of KAT, which have a well-defined

and well-studied relational semantics and deductive theory. We are able to build on this theory to provide a deductive system for program equivalence in the presence of let. Our main result is that the deductive system is complete relative to the underlying equational theory without let. The chief advantages of this approach over the related approaches mentioned above are its relative simplicity and equational completeness.

This paper is organized as follows. In Section 2, we discuss KAT and its use in program analysis. In Section 3, we define a compositional relational semantics of programs with let. In Section 4, we give a set of proof rules that allow let statements to be systematically eliminated. In Section 5, we show that the proof system is sound and complete relative to the underlying equational theory without local scoping, and provide a procedure for eliminating variable scoping expressions. By this, we do not mean that every program is equivalent to one without scoping expressions—that is not true, and a counterexample (Example 2) is given in Section 6—but rather that the equivalence of two programs with scoping expressions can be reduced to the equivalence of two programs without scoping expressions. We demonstrate the use of the proof system through several examples in Section 6.

2 Preliminary Definitions

2.1 Kleene Algebra

Kleene algebra (KA) is the algebra of regular expressions [9, 14]. The axiomatization used here is from [15]. A *Kleene algebra* is an algebraic structure $(K, +, \cdot, ^*, 0, 1)$ that satisfies the following axioms:

$$(p+q)+r = p+(q+r) \quad (2) \qquad (pq)r = p(qr) \quad (3)$$
$$p+q = q+p \quad (4) \qquad p1 = 1p = p \quad (5)$$
$$p+0 = p+p = p \quad (6) \qquad 0p = p0 = 0 \quad (7)$$
$$p(q+r) = pq+pr \quad (8) \qquad (p+q)r = pr+qr \quad (9)$$
$$1+pp^* \leq p^* \quad (10) \qquad q+pr \leq r \to p^*q \leq r \quad (11)$$
$$1+p^*p \leq p^* \quad (12) \qquad q+rp \leq r \to qp^* \leq r \quad (13)$$

This is a universal Horn axiomatization. We use pq to represent $p \cdot q$. Axioms (2)–(9) say that K is an *idempotent semiring* under $+, \cdot, 0, 1$. The adjective *idempotent* refers to the axiom $p+p=p$ (6). Axioms (10)–(13) say that p^*q is the $\leq$-least solution to $q+px \leq x$ and qp^* is the $\leq$-least solution to $q+xp \leq x$, where $\leq$ refers to the natural partial order on K defined by $p \leq q \stackrel{\text{def}}{\iff} p+q=q$.

Standard models include the family of regular sets over a finite alphabet, the family of binary relations on a set, and the family of $n \times n$ matrices over another Kleene algebra. Other more unusual interpretations include the min,+ algebra, also known as the *tropical semiring*, used in shortest path algorithms, and models consisting of convex polyhedra used in computational geometry.

There are several alternative axiomatizations in the literature, most of them infinitary. For example, a Kleene algebra is called *star-continuous* if it satisfies

the infinitary property $pq^*r = \sup_n pq^n r$. This is equivalent to infinitely many equations

$$pq^n r \leq pq^* r, \quad n \geq 0 \tag{14}$$

and the infinitary Horn formula

$$(\bigwedge_{n\geq 0} pq^n r \leq s) \to pq^* r \leq s. \tag{15}$$

All natural models are star-continuous. However, this axiom is much stronger than the finitary Horn axiomatization given above and would be more difficult to implement in an automated deduction system such as KAT-ML [1], since it would require meta-rules to handle the induction needed to establish (14) and (15).

The completeness result of [15] says that all true identities between regular expressions interpreted as regular sets of strings are derivable from the axioms. In other words, the algebra of regular sets of strings over the finite alphabet P is the free Kleene algebra on generators P. The axioms are also complete for the equational theory of relational models; that is, every equation that holds in all relational interpretations is derivable from the axioms.

See [15] for a more thorough introduction.

2.2 Kleene Algebra with Tests

A *Kleene algebra with tests* (KAT) [16] is just a Kleene algebra with an embedded Boolean subalgebra. That is, it is a two-sorted structure $(K, B, +, \cdot, {}^*, {}^-, 0, 1)$ such that

- $(K, +, \cdot, {}^*, 0, 1)$ is a Kleene algebra,
- $(B, +, \cdot, {}^-, 0, 1)$ is a Boolean algebra, and
- $B \subseteq K$.

Elements of B are called *tests*. The Boolean complementation operator ${}^-$ is defined only on tests. The axioms of Boolean algebra are purely equational. In addition to the Kleene algebra axioms above, tests satisfy the equations

$$\begin{aligned} BC &= CB & \qquad BB &= B \\ B + CD &= (B + C)(B + D) & \qquad B + 1 &= 1 \\ \overline{B + C} &= \overline{B}\ \overline{C} & \qquad \overline{BC} &= \overline{B} + \overline{C} \\ B + \overline{B} &= 1 & \qquad B\overline{B} &= 0 \\ \overline{\overline{B}} &= B \end{aligned}$$

The **while** program constructs are encoded as in propositional Dynamic Logic [10]:

$$\begin{aligned} p\,;\, q &\stackrel{\text{def}}{=} pq \\ \mathsf{if}\ B\ \mathsf{then}\ p\ \mathsf{else}\ q &\stackrel{\text{def}}{=} Bp + \overline{B}q \\ \mathsf{while}\ B\ \mathsf{do}\ p &\stackrel{\text{def}}{=} (Bp)^*\overline{B}. \end{aligned}$$

The Hoare partial correctness assertion $\{B\}\, p\, \{C\}$ is expressed as an inequality $Bp \leq pC$, or equivalently as an equation $Bp\overline{C} = 0$ or $Bp = BpC$. Intuitively, $Bp\overline{C} = 0$ says that there is no execution of p for which the input state satisfies the precondition B and the output state satisfies the postcondition $\overline{C}$, and $Bp = BpC$ says that the test C is always redundant after the execution of p under precondition B. The usual Hoare rules translate to universal Horn formulas of KAT. Under this translation, all Hoare rules are derivable in KAT; indeed, KAT is deductively complete for relationally valid propositional Hoare-style rules involving partial correctness assertions [17], whereas propositional Hoare logic is not.

See [16–18] for a more detailed introduction to KAT.

2.3 Schematic KAT

Schematic KAT (SKAT) is a specialization of KAT involving an augmented syntax to handle first-order constructs and restricted semantic actions whose intended semantics coincides with the semantics of first-order flowchart schemes over a ranked alphabet Σ [6]. Atomic actions are assignment operations $x := t$, where x is a variable and t is a Σ-term.

Five identities are paramount in proofs using SKAT:

$$x := s; y := t = y := t[x/s]; x := s \quad (y \notin FV(s)) \tag{16}$$
$$x := s; y := t = x := s; y := t[x/s] \quad (x \notin FV(s)) \tag{17}$$
$$x := s; x := t = x := t[x/s] \tag{18}$$
$$\varphi[x/t]; x := t = x := t; \varphi \tag{19}$$
$$x := x = 1 \tag{20}$$

where x and y are distinct variables and $FV(s)$ is the set of variables occurring in s in (16) and (17). The notation $s[x/t]$ denotes the result of substituting t for all occurrences of x in s. As special cases of (16) and (19), we have

$$x := s; y := t = y := t; x := s \quad (y \notin FV(s), x \notin FV(t)) \tag{21}$$
$$\varphi; x := t = x := t; \varphi \quad (x \notin FV(\varphi)) \tag{22}$$
$$s = t; x := s = s = t; x := t \tag{23}$$

In (23), $s = t$ is a test. See [6] for a more detailed development.

3 Relational Semantics

As our domain of computation we use a first-order structure $\mathfrak{A}$ of some signature Σ. A *partial valuation* is a partial map $f : \mathsf{Var} \to |\mathfrak{A}|$, where Var is a set of program variables and $|\mathfrak{A}|$ denotes the underlying set of $\mathfrak{A}$. The domain of f is denoted dom f. A stack of partial valuations is called an *environment*. Let $\sigma, \tau, \ldots$ denote environments. The notation $f :: \sigma$ denotes an environment with head f

and tail σ; thus environments grow from right to left. The empty environment is denoted ε. The *shape* of an environment $f_1 :: \cdots :: f_n$ is $\operatorname{dom} f_1 :: \cdots :: \operatorname{dom} f_n$. The *domain* of the environment $f_1 :: \cdots :: f_n$ is $\bigcup_{i=1}^{n} \operatorname{dom} f_i$. The shape of ε is ε and the domain of ε is $\varnothing$. The set of environments is denoted Env. A state of the computation is an environment, and programs will be interpreted as binary relations on environments.

In Dynamic Logic and KAT, programs are built inductively from atomic programs and tests using the regular program operators $+$, $;$, and *. In the first-order versions of these languages, atomic programs are simple assignments $x := t$, where x is a variable and t is a Σ-term. Atomic tests are atomic first-order formulas $R(t_1, \ldots, t_n)$ over the signature Σ.

To accommodate local variable scoping, we also include *let expressions* in the inductive definition of programs. A *let expression* is an expression

$$\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end} \tag{24}$$

where p is a program, the x_i are program variables, and the t_i are terms.

Operationally, when entering the scope (24), a new partial valuation is created and pushed onto the stack. The domain of this new partial valuation is $\{x_1, \ldots, x_n\}$, and the initial values of $x_1, \ldots, x_n$ are the values of $t_1, \ldots, t_n$, respectively, evaluated in the old environment. This partial valuation will be popped when leaving the scope. The locals in this partial valuation shadow any other occurrences of the same variables further down in the stack. When evaluating a variable in an environment, we search down through the stack for the first occurrence of the variable and take that value. When modifying a variable, we search down through the stack for the first occurrence of the variable and modify that occurrence. In reality, any attempt to evaluate or modify an undefined variable (one that is not in the domain of the current environment) would result in a runtime error. In the relational semantics, there would be no input-output pair corresponding to this computation.

To capture this formally in relational semantics, we use a *rebinding operator* $[x/a]$ defined on partial valuations and environments, where x is a variable and a is a value. For a partial valuation $f : \mathsf{Var} \to |\mathfrak{A}|$,

$$f[x/a](y) = \begin{cases} f(y), & \text{if } y \in \operatorname{dom} f \text{ and } y \neq x, \\ a, & \text{if } y \in \operatorname{dom} f \text{ and } y = x, \\ \text{undefined}, & \text{if } y \notin \operatorname{dom} f. \end{cases}$$

For an environment σ,

$$\sigma[x/a] = \begin{cases} f[x/a] :: \tau, & \text{if } \sigma = f :: \tau \text{ and } x \in \operatorname{dom} f, \\ f :: \tau[x/a], & \text{if } \sigma = f :: \tau \text{ and } x \notin \operatorname{dom} f, \\ \varepsilon, & \text{if } \sigma = \varepsilon. \end{cases}$$

Note that rebinding does not change the shape of the environment. In particular, $\varepsilon[x/a] = \varepsilon$. More generally, if $x \in \operatorname{dom} f$ for any partial valuation in σ, then $\sigma[x/a] = \sigma$.

The value of a variable x in an environment σ is

$$\sigma(x) = \begin{cases} f(x), & \text{if } \sigma = f :: \tau \text{ and } x \in \operatorname{dom} f, \\ \tau(x), & \text{if } \sigma = f :: \tau \text{ and } x \notin \operatorname{dom} f, \\ \text{undefined}, & \text{if } \sigma = \varepsilon. \end{cases}$$

The value of a term t in an environment σ is defined inductively on t in the usual way. Note that $\sigma(t)$ is defined iff $x \in \operatorname{dom} \sigma$ for all x occurring in t.

A program is interpreted as a binary relation on environments. The binary relation associated with p is denoted $[\![p]\!]$. The semantics of assignment is

$$[\![x := t]\!] = \{(\sigma, \sigma[x/\sigma(t)]) \mid \sigma(t) \text{ and } \sigma(x) \text{ are defined}\}.$$

Note that both x and t must be defined by σ for there to exist an input-output pair with first component σ.

The semantics of scoping is

$$\begin{aligned} &[\![\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}]\!] \\ &= \{(\sigma, \mathsf{tail}(\tau)) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } (f :: \sigma, \tau) \in [\![p]\!]\}, \end{aligned} \tag{25}$$

where f is the environment such that $f(x_i) = \sigma(t_i)$, $1 \le i \le n$.

As usual with binary relation semantics, the semantics of the regular program operators $+$, $;$, and * are union, relational composition, and reflexive transitive closure, respectively. For an atomic test $R(t_1, \ldots, t_n)$,

$$\begin{aligned} &[\![R(t_1, \ldots, t_n)]\!] \\ &= \{(\sigma, \sigma) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } \mathfrak{A}, \sigma \vDash R(t_1, \ldots, t_n)\}. \end{aligned}$$

where $\vDash$ is satisfaction in the usual sense of first-order logic. The Boolean operator ! (weak negation) is defined on atomic formulas by

$$\begin{aligned} &[\![!R(t_1, \ldots, t_n)]\!] \\ &= \{(\sigma, \sigma) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } \mathfrak{A}, \sigma \vDash \neg R(t_1, \ldots, t_n)\}. \end{aligned}$$

Because of the definedness condition, this is not the same as classical negation $\neg$, which we need in order to use the axioms of Kleene algebra with tests. However, classical negation can be obtained from weak negation and the ability to check whether a variable is undefined. That is, we must have a test $\mathsf{undefined}(x)$ with semantics

$$[\![\mathsf{undefined}(x)]\!] = \{(\sigma, \sigma) \mid \sigma(x) \text{ is undefined}\}.$$

This is a very reasonable assumption. Even without this capability, the short-circuiting Boolean operators can be defined by

$$\begin{aligned} [\![\varphi \mathbin{\&\&} \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\ [\![\varphi \mathbin{||} \psi]\!] &= [\![\varphi]\!] \cup ([\![!\varphi]\!] \cap [\![\psi]\!]) \\ [\![!(\varphi \mathbin{\&\&} \psi)]\!] &= [\![!\varphi]\!] \cup ([\![\varphi]\!] \cap [\![!\psi]\!]) \quad = \quad [\![!\varphi \mathbin{||} !\psi]\!] \\ [\![!(\varphi \mathbin{||} \psi)]\!] &= [\![!\varphi]\!] \cap [\![!\psi]\!] \quad = \quad [\![!\varphi \mathbin{\&\&} !\psi]\!] \\ [\![!!\varphi]\!] &= [\![\varphi]\!]. \end{aligned}$$

These definitions give a relational semantics for the familiar short-circuiting Boolean operators &&, ||, and ! in languages such as C and Java.

Example 1. Consider the program

$$\begin{array}{ll}\mathsf{let} & x = 1 \\ \mathsf{in} & x := y + z; \\ & \mathsf{let}\ y = x + 2\ \mathsf{in}\ y := y + z; z := y + 1\ \mathsf{end}; \\ & y := x \\ \mathsf{end}. & \end{array}$$

Say we start in state $(\mathtt{y} = 5, \mathtt{z} = 20)$. Here are the successive states of the computation:

After ...	*the state is ...*
entering the outer scope	$(\mathtt{x} = 1) :: (\mathtt{y} = 5, \mathtt{z} = 20)$
executing the first assignment	$(\mathtt{x} = 25) :: (\mathtt{y} = 5, \mathtt{z} = 20)$
entering the inner scope	$(\mathtt{y} = 27) :: (\mathtt{x} = 25) :: (\mathtt{y} = 5, \mathtt{z} = 20)$
executing the next assignment	$(\mathtt{y} = 47) :: (\mathtt{x} = 25) :: (\mathtt{y} = 5, \mathtt{z} = 20)$
executing the next assignment	$(\mathtt{y} = 47) :: (\mathtt{x} = 25) :: (\mathtt{y} = 5, \mathtt{z} = 48)$
exiting the inner scope	$(\mathtt{x} = 25) :: (\mathtt{y} = 5, \mathtt{z} = 48)$
executing the last assignment	$(\mathtt{x} = 25) :: (\mathtt{y} = 25, \mathtt{z} = 48)$
exiting the outer scope	$(\mathtt{y} = 25, \mathtt{z} = 48)$

Lemma 1. *If $(\sigma, \tau) \in [\![p]\!]$, then σ and τ have the same shape.*

Proof. This is true of the assignment statement and preserved by all program operators. □

The goal of presenting a semantics for a language with local state is to allow reasoning about programs without the need for context. A *context* $C[\text{-}]$ is just a program expression with a distinguished free program variable. Relational semantics captures all contextual information in the state, thus making contexts superfluous in program equivalence arguments. This is reflected in the following theorem.

Theorem 1. *For program expressions p and q, $[\![C[p]]\!] = [\![C[q]]\!]$ for all contexts $C[\text{-}]$ iff $[\![p]\!] = [\![q]\!]$.*

This is a special case of a result proved in more generality in [2]. The direction ($\rightarrow$) is immediate by taking $C[\text{-}]$ to be the trivial context consisting of a single program variable. The reverse direction follows from an inductive argument, observing that the semantics is fully compositional, the semantics of a compound expression being completely determined by the semantics of its subexpressions.

4 Axioms and Basic Properties

In this section we present a set of axioms that can be used to systematically eliminate all local scopes, allowing us to reduce the equivalence problem to equivalence

in the traditional "flat" semantics in which all variables are global. Although the relational semantics presented in Section 3 is a special case of the semantics presented in [2] for higher-order programs, an axiomatization was not considered in that work.

Axioms

A. If the y_i are distinct and do not occur in p, $1 \leq i \leq n$, then the following two programs are equivalent:

$$\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}$$
$$\mathsf{let}\ y_1 = t_1, \ldots, y_n = t_n\ \mathsf{in}\ p[x_i/y_i \mid 1 \leq i \leq n]\ \mathsf{end}$$

where $p[x_i/y_i \mid 1 \leq i \leq n]$ refers to the simultaneous substitution of y_i for all occurrences of x_i in p, $1 \leq i \leq n$, including bound occurrences and those on the left-hand sides of assignments. This transformation is known as *α-conversion*.

B. If y does not occur in s and y and x are distinct, then the following two programs are equivalent:

$$\mathsf{let}\ x = s\ \mathsf{in}\ \mathsf{let}\ y = t\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$
$$\mathsf{let}\ y = t[x/s]\ \mathsf{in}\ \mathsf{let}\ x = s\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$

In particular, the following two programs are equivalent, provided x and y are distinct, x does not occur in t, and y does not occur in s:

$$\mathsf{let}\ x = s\ \mathsf{in}\ \mathsf{let}\ y = t\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$
$$\mathsf{let}\ y = t\ \mathsf{in}\ \mathsf{let}\ x = s\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$

C. If x does not occur in s, then the following two programs are equivalent:

$$\mathsf{let}\ x = s\ \mathsf{in}\ \mathsf{let}\ y = t\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$
$$\mathsf{let}\ x = s\ \mathsf{in}\ \mathsf{let}\ y = t[x/s]\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$

This holds even if x and y are the same variable.

D. If x_1 does not occur in $t_2, \ldots, t_n$, then the following two programs are equivalent:

$$\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}$$
$$\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \mathsf{let}\ x_2 = t_2, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$$

E. If t is a closed term (no occurrences of variables), then the following two programs are equivalent:

$$\mathsf{skip} \qquad \mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{skip}\ \mathsf{end}$$

where skip is the identity function on states.

F. If x does not occur in pr, then the following two programs are equivalent:

$$p; \mathsf{let}\ x = t\ \mathsf{in}\ q\ \mathsf{end}; r \qquad \mathsf{let}\ x = t\ \mathsf{in}\ pqr\ \mathsf{end}$$

G. If x does not occur in p and t is closed, then the following two programs are equivalent:

$$p + \mathsf{let}\ x = t\ \mathsf{in}\ q\ \mathsf{end} \qquad \mathsf{let}\ x = t\ \mathsf{in}\ p + q\ \mathsf{end}$$

The proviso "t is closed" is necessary: if value of t is initially undefined, then the program on the left may halt, whereas the program on the right never does.

H. If x does not occur in t, then the following two programs are equivalent:

$$(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^* \qquad \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^*\ \mathsf{end}$$

where a is any closed term. The proviso that x not occur in t is necessary, as the following counterexample shows. Take $t = x$ and p the assignment $y := a$. The program on the right contains the pair $(y = b, y = a)$ for $b \neq a$, whereas the program on the left does not, since x must be defined in the environment in order for the starred program to be executed once.

I. If x does not occur in t and a is a closed term, then the following two programs are equivalent:

$$\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \qquad \mathsf{let}\ x = a\ \mathsf{in}\ x := t; p\ \mathsf{end}$$

J. If x does not occur in t, then the following two programs are equivalent:

$$\mathsf{let}\ x = s\ \mathsf{in}\ p\ \mathsf{end}; x := t \qquad x := s; p; x := t$$

Theorem 2. *Axioms A–J are sound with respect to the binary relation semantics of Section 3.*

Proof. Most of the arguments are straightforward relational reasoning. Perhaps the least obvious is Axiom H, which we argue explicitly. Suppose that x does not occur in t. Let a be any closed term. We wish to show that the following two programs are equivalent:

$$(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^* \qquad \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^*\ \mathsf{end}$$

Extend the nondeterministic choice operator to infinite sets in the obvious way. This is possible because infinite sums exist in relational interpretations. We have

$$(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^* = \sum_n (\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^n$$

$$\mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^*\ \mathsf{end} = \mathsf{let}\ x = a\ \mathsf{in}\ \sum_n (x := t; p)^n\ \mathsf{end}$$

$$= \sum_n \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n\ \mathsf{end}$$

the last by a straightforward infinitary generalization of Axiom G, which holds in relational models. It therefore suffices to prove that for any n,

$$(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^n = \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n\ \mathsf{end}$$

This is true for $n = 0$ by Axiom E. Now suppose it is true for n. Then

$$\begin{aligned}
&(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^{n+1} \\
&= (\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^n; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \qquad (26) \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n; x := t; p\ \mathsf{end} \qquad (27) \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^{n+1}\ \mathsf{end}
\end{aligned}$$

where (26) follows from the induction hypothesis and (27) follows from the identity

$$\mathsf{let}\ x = a\ \mathsf{in}\ q\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \quad = \quad \mathsf{let}\ x = a\ \mathsf{in}\ q; x := t; p\ \mathsf{end} \qquad (28)$$

To justify (28), observe that since x does not occur in t by assumption, p is executed in exactly the same environment on both sides of the equation.

When proving programs equivalent, it is helpful to know we can permute local variable declarations and remove unnecessary ones.

Lemma 2.

(i) *For any permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, the following two programs are equivalent:*

$$\begin{aligned}
&\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end} \\
&\mathsf{let}\ x_{\pi(1)} = t_{\pi(1)}, \ldots, x_{\pi(n)} = t_{\pi(n)}\ \mathsf{in}\ p\ \mathsf{end}.
\end{aligned}$$

(ii) *If x does not occur in p, and if t is a closed term, then the following two programs are equivalent:*

$$p \qquad \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end}.$$

The second part of Lemma 2 is similar to the first example of Meyer and Sieber [23] in which a local variable unused in a procedure call can be eliminated.

5 Flattening

To prove equivalence of two programs p, q with scoping, we transform the programs so as to remove all scoping expressions to obtain two transformed programs p', q', then prove the equivalence of p', q'. It is important to note that the transformed program p' is not equivalent to the original program p in general. However, p', q' are equivalent in the "flat" semantics iff p, q were equivalent in the semantics of Section 3. Thus the process is complete modulo the theory of programs without scope. The transformations are applied in the following stages.

Step 1 Apply α-conversion (Axiom A) to both programs to make all bound variables unique. This is done from the innermost scopes outward. In particular, no bound variable in the first program appears in the second program and vice-versa. The resulting programs are equivalent to the originals.

Step 2 Let $x_1, \ldots, x_n$ be any list of variables containing all bound variables that occur in either program after Step 1. Use the transformation rules of Axioms A–J to convert the programs to the form $\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ p\ \mathsf{end}$ and $\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ q\ \mathsf{end}$, where p and q do not have any scoping expressions and a is a closed term. The scoping expressions can be moved outward using Axioms F–H. Adjacent scoping expressions can be combined using Axioms C and D. Finally, all bindings can be put into the form $x = a$ using Axiom I.

Step 3 Now for p, q with no scoping and a a closed term, the two programs

$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ p\ \mathsf{end}$$
$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ q\ \mathsf{end}$$

are equivalent iff the two programs

$$x_1 := a; \cdots; x_n := a; p; x_1 := a; \cdots; x_n := a$$
$$x_1 := a; \cdots; x_n := a; q; x_1 := a; \cdots; x_n := a$$

are equivalent with respect to the "flat" binary relation semantics in which states are just partial valuations. We have shown

Theorem 3. *Axioms A–J of Section 4 are sound and complete for program equivalence relative to the underlying equational theory without local scoping.*

6 Examples

We demonstrate the use of the axiom system through several examples. The first example proves that two versions of a program to swap the values of two variables are equivalent when the domain of computation is the integers in binary representation.

Example 2. The following two programs are equivalent:

$$\begin{array}{lll} \mathsf{let} & t = x & x := x \oplus y; \\ \mathsf{in} & x := y; & y := x \oplus y; \\ & y := t & x := x \oplus y \\ \mathsf{end} & & \end{array}$$

where $\oplus$ is the bitwise xor operator. The first program uses a local variable to store the value of x temporarily. The second program does not need a temporary value; it uses xor to switch the bits in place. Without the ability to handle local variables, it would be impossible to prove these two programs equivalent, because

the first program includes an additional variable t. In general, without specific information about the domain of computation and without an operator like $\oplus$, it would be impossible to prove the left-hand program equivalent to any let-free program.

Proof. We apply Axiom I to the first program and Lemma 2 to the second program to get

$$\begin{array}{ll} \mathsf{let} & t = a \\ \mathsf{in} & t := x; \\ & x := y; \\ & y := t \\ \mathsf{end}. & \end{array} \qquad\qquad \begin{array}{ll} \mathsf{let} & t = a \\ \mathsf{in} & x := x \oplus y; \\ & y := x \oplus y; \\ & x := x \oplus y \\ \mathsf{end} & \end{array}$$

respectively, where a is a closed term. From Theorem 3, it suffices to show the following programs are equivalent:

$$\begin{array}{l} t := a; \\ t := x; \\ x := y; \\ y := t; \\ t := a \end{array} \qquad\qquad \begin{array}{l} t := a; \\ x := x \oplus y; \\ y := x \oplus y; \\ x := x \oplus y; \\ t := a. \end{array}$$

We have reduced the problem to an equation between let-free programs. The remainder of the argument is a straightforward application of the axioms of schematic KAT [6] and the properties of the domain of computation. See Appendix A for the rest of the proof. □

The second example shows that a local variable in a loop need only be declared once if the variable's value is not changed by the body of the loop.

Example 3. If the final value of x after exectuing program p is always a, that is, if p is equivalent to $p;(x = a)$ for closed term a, then the following two programs are equivalent:

$$(\mathsf{let}\ x = a\ \mathsf{in}\ p\ \mathsf{end})^* \qquad\qquad \mathsf{let}\ x = a\ \mathsf{in}\ p^*\ \mathsf{end}.$$

Proof. First, we use Axiom H to convert the program on the left-hand side to

$$\mathsf{let}\ x = a\ \mathsf{in}\ (x := a; p)^*\ \mathsf{end}.$$

It suffices to show the following flattened programs are equivalent:

$$x := a; (x := a; p)^*; x := a \qquad\qquad x := a; p^*; x := a.$$

The equivalence follows from basic theorems of KAT and our assumption $p = p;(x = a)$. See Appendix A for the rest of the proof. □

The next example is important in path-sensitive analysis for compilers. It shows that a program with multiple conditionals all guarded by the same test needs only one local variable for operations in both branches of the conditionals.

Example 4. If x and w do not occur in p and the program $(y = a); p$ is equivalent to the program $p; (y = a)$ (that is, the execution of p does not affect the truth of the test $y = a$), then the following two programs are equivalent:

$$\begin{array}{l}\mathsf{let}\ \ x = 0, w = 0\\ \mathsf{in}\ \ \ (\mathsf{if}\ y = a\ \mathsf{then}\ x := 1\ \mathsf{else}\ w := 2); p; \mathsf{if}\ y = a\ \mathsf{then}\ y := x\ \mathsf{else}\ y := w\\ \mathsf{end}\end{array}$$

$$\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ (\mathsf{if}\ y = a\ \mathsf{then}\ x := 1\ \mathsf{else}\ x := 2); p; y := x\\ \mathsf{end}.\end{array}$$

Proof. First we note that it follows purely from reasoning in KAT that $(y = a); p$ is equivalent to $(y = a); p; (y = a)$ and that $(y \neq a); p$ is equivalent to $p; (y \neq a)$ and also to $(y \neq a); p; (y \neq a)$.

We use laws of distributivity and Boolean tests from KAT and our assumptions to transform the first program into

$$\begin{array}{l}\mathsf{let}\ \ x = 0, w = 0\\ \mathsf{in}\ \ \ (y = a; x := 1; p; y = a; y := x) + (y \neq a; w := 2; p; y \neq a; y := w)\\ \mathsf{end}.\end{array}$$

Axiom D allows us to transform this program into

$$\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ \mathsf{let}\ \ w = 0\\ \qquad \mathsf{in}\ \ \ (y = a; x := 1; p; y = a; y := x) + (y \neq a; w := 2; p; y \neq a; y := w)\\ \qquad \mathsf{end}\\ \mathsf{end}.\end{array}$$

By two applications of Axiom G, we get

$$\left(\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ y = a; x := 1; p; y = a; y := x\\ \mathsf{end}\end{array}\right) + \left(\begin{array}{l}\mathsf{let}\ \ w = 0\\ \mathsf{in}\ \ \ y \neq a; w := 2; p; y \neq a; y := w\\ \mathsf{end}\end{array}\right).$$

Using α-conversion (Axiom A) to replace w with x, this becomes

$$\left(\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ y = a; x := 1; p; y = a; y := x\\ \mathsf{end}\end{array}\right) + \left(\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ y \neq a; x := 2; p; y \neq a; y := x\\ \mathsf{end}\end{array}\right).$$

This program is equivalent to

$$\begin{array}{l}\mathsf{let}\ \ x = 0\\ \mathsf{in}\ \ \ (y = a; x := 1; p; y = a; y := x) + (y \neq a; x := 2; p; y \neq a; y := x)\\ \mathsf{end}.\end{array}$$

by a simple identity

$$\mathsf{let}\ x = a\ \mathsf{in}\ p + q\ \mathsf{end} \quad = \quad \mathsf{let}\ x = a\ \mathsf{in}\ p\ \mathsf{end} + \mathsf{let}\ x = a\ \mathsf{in}\ q\ \mathsf{end}.$$

It is easy to see that this identity is true, as both p and q are executed in the same state on both sides of the equation. It can also be justified axiomatically using Axioms A, D, and G and a straightforward application of Theorem 3.

Finally, we use laws of distributivity and Booleans to get

$$\begin{array}{ll} \mathsf{let} & x = 0 \\ \mathsf{in} & (\mathsf{if}\ y = a\ \mathsf{then}\ x := 1\ \mathsf{else}\ x := 2); p; y := x \\ \mathsf{end}. & \end{array}$$

which is what we wanted to prove. □

7 Conclusion

We have presented a relational semantics for first-order programs with a let construct for local variable scoping and a set of equational axioms for reasoning about program equivalence in this language. The axiom system allows the let construct to be systematically eliminated, thereby reducing the equivalence arguments to the let-free case. This system admits algebraic equivalence proofs for programs with local variables in the equational style of schematic KAT. We have given several examples that illustrate that in many cases, it is possible to reason purely axiomatically about programs with local variables without resorting to semantic arguments involving heaps, pointers, or other complicated semantic constructs.

Acknowledgments

We would like to thank Matthew Fluet, Riccardo Pucella, Sigmund Cherem, and the anonymous referees for their valuable input. This work was supported by the National Science Foundation under grant CCF-0635028. Any views or conclusions expressed herein are those of the authors and do not necessarily represent the official policies or endorsements of the National Science Foundation or the United States government.

References

1. Kamal Aboul-Hosn and Dexter Kozen. KAT-ML: An interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–33, 2006.
2. Kamal Aboul-Hosn and Dexter Kozen. Relational semantics for higher-order programs. In Tarmo Uustalu, editor, *Proc. 8th Int. Conf. Mathematics of Program Construction (MPC'06)*, volume 4014 of *Lecture Notes in Computer Science*, pages 29–48. Springer, July 2006.
3. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 334, Washington, DC, USA, 1998. IEEE Computer Society.

4. Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.
5. Samson Abramsky and Guy McCusker. Call-by-value games. In Mogens Nielsen and Wolfgang Thomas, editors, *CSL*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1997.
6. Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.
7. R. J. R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Non deterministic Programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness - REX Workshop*, volume 430, pages 42–66, Berlin, Germany, 1989. Springer-Verlag.
8. Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 328–342, Albequerque, New Mexico, 1992.
9. John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.
10. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.
11. T. S. E. Maibaum Gabriel Baum, Marcelo F. Frias. A logic for real-time systems specification, its algebraic semantics, and equational calculus. In *Proc. Conf. Algebraic Methodology and Software Technology (AMAST'98)*, pages 91–105, 1998.
12. Joseph Y. Halpern, Albert R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free?(preliminary report). In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 245–257, New York, NY, USA, 1984. ACM Press.
13. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
14. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.
15. Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
16. Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
17. Dexter Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.
18. Dexter Kozen and Jerzy Tiuryn. Substructural logic and partial correctness. *Trans. Computational Logic*, 4(3):355–378, July 2003.
19. James Laird. A game semantics of local names and good variables. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2004.
20. I. A. Mason and C. L. Talcott. Axiomatizing operational equivalence in the presence of effects. In *Proc. 4th Symp. Logic in Computer Science (LICS'89)*, pages 284–293. IEEE, 1989.
21. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

22. I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on Logic in Computer Science*, pages 186–197. IEEE, 1992.
23. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Proc. 15th Symposium on Principles of Programming Languages (POPL'88)*, pages 191–203, New York, NY, USA, 1988. ACM Press.
24. Robert Milne and C. Strachey. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 1977.
25. Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1998.
26. P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In P. T. Johnstone M. P. Fourman and A. M. Pitts, editors, *Applications of Categories in Computer Science*, L.M.S. Lecture Note Series, pages 217–238. Cambridge University Press, 1992.
27. Frank Joseph Oles. *A category-theoretic approach to the semantics of programming languages*. PhD thesis, Syracuse University, 1982.
28. P. Pepper. A simple calculus of program transformations (inclusive of induction). *Science of Computer Programming*, 9(3):221–262, 1987.
29. A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.
30. A. M. Pitts. Operational semantics and program equivalence. Technical report, INRIA Sophia Antipolis, 2000. Lectures at the International Summer School On Applied Semantics, APPSEM 2000, Caminha, Minho, Portugal, 9-15 September 2000.
31. A. M. Pitts and I. D. B. Stark. Operational reasoning in functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.
32. Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, August 1993.
33. G. Plotkin. Full abstraction, totality and PCF, 1997.
34. J.C. Reyonlds. The essence of ALGOL. In J.W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, North-Holland, Amsterdam, 1981.
35. D.S. Scott. Mathematical concepts in programmng language semantics. In *Proc. 1972 Spring Joint Computer Conferences*, pages 225–34, Montvale, NJ, 1972. AFIPS Press.
36. Ian Stark. Categorical models for local names. *LISP and Symbolic Computation*, 9(1):77–107, February 1996.
37. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981.

A Examples

Example 2. In the main text (Section 6, Example 2), we reduced the problem to an equation between let-free programs. The remainder of the argument is a straightforward application of the axioms of schematic KAT and the properties of the domain of computation. Fragments of the statements that changed from one step to the next are in bold.

Using 18, the right-hand side is equivalent to

$$\begin{aligned}
&t := a;\\
&x := x \oplus y;\\
&y := x \oplus y;\\
&x := x \oplus y;\\
&\boldsymbol{t := y;}\\
&\boldsymbol{t := a}.
\end{aligned}$$

By (16), this is equivalent to

$$\begin{aligned}
&t := a;\\
&x := x \oplus y;\\
&y := x \oplus y;\\
&\boldsymbol{t := y;}\\
&\boldsymbol{x := x \oplus y;}\\
&t := a.
\end{aligned}$$

By (16), this is equivalent to

$$\begin{aligned}
&t := a;\\
&x := x \oplus y;\\
&\boldsymbol{t := x \oplus y;}\\
&\boldsymbol{y := x \oplus y;}\\
&x := x \oplus y;\\
&t := a.
\end{aligned}$$

By (17), this is equivalent to

$$\begin{aligned}
&t := a;\\
&x := x \oplus y;\\
&t := x \oplus y;\\
&\boldsymbol{y := t;}\\
&x := x \oplus y;\\
&t := a.
\end{aligned}$$

By (16), this is equivalent to

$$\begin{aligned}
&t := a;\\
&x := x \oplus y;\\
&t := x \oplus y;\\
&\boldsymbol{x := x \oplus t;}\\
&\boldsymbol{y := t;}\\
&t := a.
\end{aligned}$$

By (17) and the fact that $x \oplus x \oplus y = y$, this is equivalent to

$$\begin{array}{l} t := a; \\ x := x \oplus y; \\ t := x \oplus y; \\ \boldsymbol{x := y;} \\ y := t; \\ t := a. \end{array}$$

By (16) and the fact that $x \oplus y \oplus y = x$, this is equivalent to

$$\begin{array}{l} t := a; \\ \boldsymbol{t := x;} \\ \boldsymbol{x := x \oplus y;} \\ x := y; \\ y := t; \\ t := a. \end{array}$$

Finally, by (18), this is equivalent to

$$\begin{array}{l} t := a; \\ t := x; \\ x := y; \\ y := t; \\ t := a, \end{array}$$

which is the left-hand side with which we started. □

Example 3. In the main text (Section 6, Example 3), we reduced the problem to showing the equivalence of the following two flattened programs:

$$x := a; (x := a; p)^*; x := a \qquad\qquad x := a; p^*; x := a.$$

The equivalence follows from basic theorems of KAT and our assumption $p = p; (x = a)$. Changes from step to step are in bold.

Using (18) on the left-hand side gives us

$$x := a; (x := a; p)^*; \boldsymbol{x := a; x := a}.$$

We next use the sliding rule, $x; (y; x)^* = (x; y)^*; x$ to get

$$x := a; \boldsymbol{x := a; (p; x := a)^*;} x := a.$$

See [15] for more information about the sliding rule. Again applying (18), we get

$$\boldsymbol{x := a;} (p; x := a)^*; x := a.$$

Applying our assumption $p = p; (x = a)$ yields

$$x := a; (p; \boldsymbol{(x = a);} x := a)^*; x := a.$$

Using (23) and (20) gives us

$$x := a; (p; (x = a); \mathbf{1})^*; x := a.$$

Finally, using (5) and our assumption gives us

$$x := a; p^*; x := a,$$

which is what we wanted. □