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. Most previous work on the semantics of programs with local state involves complex storage modeling with pointers and memory cells, complicated categorical constructions, or reasoning in the presence of context. In this paper, we explore the extent to which relational semantics and axiomatic reasoning in the style of Kleene algebra can be used to avoid these complications. We provide (i) a fully compositional relational semantics for a first-order programming language with a construct for local variable scoping; and (ii) an equational proof system based on Kleene algebra with tests for proving equivalence of programs in this language. We show that the proof system is sound and complete relative to the underlying equational theory without local scoping. We illustrate the use of the system with several examples.

1 Introduction

Reasoning about programs with local state is an important and difficult problem that has attracted much attention over the years. Most previous work involves complex storage modeling with pointers and memory cells or complicated categorical constructions to capture the intricacies of programming with state. Reasoning about the equality of such programs typically involves the notion of *contextual* or *observable equivalence*, where two programs are considered equivalent if either can be put in the context of a larger program and yield the same value. Pitts [1] explains that these notions are difficult to define formally, because there is no clear agreement on the meaning of *program context* and *observable behavior*. A common goal is to design a semantics that is *fully abstract*, where observable equivalence implies semantic equivalence, although this notion makes the most sense in a purely functional context (see for example [2, 3]).

Seminal work by Meyer and Sieber [4] introduced a framework for proving the equivalence of ALGOL procedures with no parameters. Much attention has focused on the use of denotational semantics to model a set of storage locations [5–8]. The inability to prove some simple program equivalences using traditional techniques led several researchers to take a categorical approach [9–11]. See [12] for more information regarding the history of these approaches.

More recently, several researchers have investigated the use of operational semantics to reason about ML programs with references. While operational semantics can be easier to understand, their use makes reasoning about programs more complex. Mason and Talcott [13–15] considered a λ-calculus extended with state operations. By defining axioms in the form of contextual assertions, Mason and Talcott were able to prove the equivalence in several examples of Meyer and Sieber. Pitts and Stark [1, 16–18] also use operational semantics.

Others have looked at using game semantics to reason about programs with local state [19–22]. Several full abstraction results have come from using game semantics to represent languages with state.

In [23], we presented a fully compositional relational semantics for higher-order programs, and showed how it could be used to avoid intricate memory modeling and the explicit use of context in program equivalence proofs. We showed how to handle several examples of Meyer and Sieber [4] in our framework. However, in that paper, we did not attempt to formulate an equational axiomatization; all arguments were based on semantic reasoning.

In this paper we consider a restricted language without higher-order programs but with a *let* construct for declaring local variables with limited scope:

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

In the presence of higher-order programs, this construct can be encoded as a λ-term $(\lambda x.p)t$, but here we take (1) as primitive. The standard relational semantics used in first-order KAT and Dynamic Logic 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.

This semantics is a restriction of the relational semantics of [23] for interpreting higher-order programs. There, instead of a stack, we used a more complicated tree-like structure called a *closure structure*. Nevertheless, it is worthwhile giving an explicit treatment of this important special case. The *let* construct interacts relatively 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* that is complete relative to the underlying equational theory without *let*.

This paper is organized as follows. In Section 2, we define a compositional relational semantics of programs with *let*. In Section 3, we give a set of proof rules that allow *let* statements to be systematically eliminated. In Section 4, 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 "eliminating variable scoping expressions," we do not mean that every program is equivalent to one without scoping expressions—that is not true, and a counterexample is given in Section 5—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 5.

2 Relational Semantics

The *domain of computation* is 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. 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 dom $f_1 :: \cdots ::$ dom f_n. The *domain* of the environment $f_1 :: \cdots :: f_n$ is $\bigcup_{i=1}^{n}$ 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

$$\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end} \tag{2}$$

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

Operationally, when entering the scope (2), 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 $f : \mathsf{Var} \to |\mathfrak{A}|$,

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

For an environment σ,

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

Note that rebinding does not change the shape of the environment. In particular, $\varepsilon[x/a] = \varepsilon$.

The value of a variable x in an environment σ is

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

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 $x \in \text{dom } \sigma$ 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

$$\begin{aligned} &[\![\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}]\!] \\ &= \{(\sigma, \mathsf{tail}(\tau)) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } (f :: \sigma, \tau) \in [\![p]\!]\}, \end{aligned} \tag{3}$$

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

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

This is not the same as classical negation $\neg$, which we need in order to use the axioms of Kleene algebra with tests. However, in the presence of !, classical negation is tantamount to the ability to check whether a variable is undefined. That is, we must have a test $\mathsf{undefined}(x)$ with semantics

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

Example 1. Consider the program

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

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 [23]. 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.

3 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 2 is a special case of the semantics presented in [23] 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:

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

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:

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

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:

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

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

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

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:

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

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 2.*

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:

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

Extending the nondeterministic choice operator to infinite sets in the obvious way, 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. 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

$$\begin{aligned}
&(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^{n+1} \\
&= (\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^n; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \qquad (4) \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n; x := t; p\ \mathsf{end} \qquad (5) \\
&= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^{n+1}\ \mathsf{end}
\end{aligned}$$

where (4) follows from the induction hypothesis and (5) follows from the identity

$$\mathsf{let}\ x = a\ \mathsf{in}\ q\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \quad = \quad \mathsf{let}\ x = a\ \mathsf{in}\ q; x := t; p\ \mathsf{end} \qquad (6)$$

To justify (6), 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 [4] in which a local variable unused in a procedure call can be eliminated.

4 Flattening

To prove equivalence of two programs p, q with scoping, we transform the programs so as to remove all scoping expressions, then prove the equivalence of the two resulting programs. The transformed programs are equivalent to the original ones except for the last step. The two transformed programs are equivalent in the "flat" semantics iff the original ones were equivalent in the semantics of Section 2. 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 3 are sound and complete for program equivalence relative to the underlying equational theory without local scoping.*

5 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.

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 Lemma 2 to convert the second program to

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

where a is a closed term. Next, we apply Axiom I to the first program to get

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

From Theorem 3, it suffices to show the following programs are equivalent:

$$\begin{array}{ll} t := a; & t := a; \\ t := x; & x := x \oplus y; \\ x := y; & y := x \oplus y; \\ y := t; & x := x \oplus y; \\ t := a & t := a \end{array}$$

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 [24] and the properties of the domain of computation. □

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:

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

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:

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

The equivalence follows from basic theorems of KAT and our assumption $p = p;(x = a)$. □

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

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

which is what we wanted to prove. □

6 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.

References

1. Pitts, A.M.: 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, September 2000.
2. Plotkin, G.: Full abstraction, totality and PCF (1997)
3. Cartwright, R., Felleisen, M.: Observable sequentiality and full abstraction. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albequerque, New Mexico (1992) 328–342
4. Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: Proc. 15th Symposium on Principles of Programming Languages (POPL'88), New York, NY, USA, ACM Press (1988) 191–203
5. Milne, R., Strachey, C.: A Theory of Programming Language Semantics. Halsted Press, New York, NY, USA (1977)
6. Scott, D.: Mathematical concepts in programmng language semantics. In: Proc. 1972 Spring Joint Computer Conferneces, Montvale, NJ, AFIPS Press (1972) 225–34

7. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA (1981)
8. Halpern, J.Y., Meyer, A.R., Trakhtenbrot, B.A.: 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, New York, NY, USA, ACM Press (1984) 245–257
9. Stark, I.: Categorical models for local names. LISP and Symbolic Computation **9**(1) (1996) 77–107
10. Reyonlds, J.: The essence of ALGOL. In de Bakker, J., van Vliet, J.C., eds.: Algorithmic Languages, North-Holland, Amsterdam (1981) 345–372
11. Oles, F.J.: A category-theoretic approach to the semantics of programming languages. PhD thesis, Syracuse University (1982)
12. O'Hearn, P.W., Tennent, R.D.: Semantics of local variables. In M. P. Fourman, P.T.J., Pitts, A.M., eds.: Applications of Categories in Computer Science. L.M.S. Lecture Note Series, Cambridge University Press (1992) 217–238
13. Mason, I.A., Talcott, C.L.: Axiomatizing operational equivalence in the presence of effects. In: Proc. 4th Symp. Logic in Computer Science (LICS'89), IEEE (1989) 284–293
14. Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. Journal of Functional Programming **1** (1991) 287–327
15. Mason, I.A., Talcott, C.L.: References, local variables and operational reasoning. In: Seventh Annual Symposium on Logic in Computer Science, IEEE (1992) 186–197
16. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or what's new? In Borzyszkowski, A.M., Sokolowski, S., eds.: MFCS. Volume 711 of Lecture Notes in Computer Science., Springer (1993) 122–141
17. Pitts, A.M.: Operationally-based theories of program equivalence. In Dybjer, P., Pitts, A.M., eds.: Semantics and Logics of Computation. Publications of the Newton Institute. Cambridge University Press (1997) 241–298
18. Pitts, A.M., Stark, I.D.B.: Operational reasoning in functions with local state. In Gordon, A.D., Pitts, A.M., eds.: Higher Order Operational Techniques in Semantics. Cambridge University Press (1998) 227–273
19. Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. In: LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (1998) 334–344
20. Laird, J.: A game semantics of local names and good variables. In Walukiewicz, I., ed.: FoSSaCS. Volume 2987 of Lecture Notes in Computer Science., Springer (2004) 289–303
21. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions. Electr. Notes Theor. Comput. Sci. **3** (1996)
22. Abramsky, S., McCusker, G.: Call-by-value games. In Nielsen, M., Thomas, W., eds.: CSL. Volume 1414 of Lecture Notes in Computer Science., Springer (1997) 1–17
23. Aboul-Hosn, K., Kozen, D.: Relational semantics for higher-order programs. In: Proc. Mathematics of Program Construction (MPC06). (2006) To appear.
24. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report 2001-1844, Computer Science Department, Cornell University (2001)