Relational Semantics of Local Variable Scoping

Kamal Aboul-Hosn
kamal@cs.cornell.edu

Dexter Kozen
kozen@cs.cornell.edu

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

July 18, 2005

Abstract

Most previous work on the equivalence of programs in the presence of local state has involved intricate memory modeling and the notion of contextual (observable) equivalence. We show how relational semantics can be used to avoid these complications. We define a notion of local variable scoping, along with a purely compositional semantics based on binary relations, such that all contextual considerations are completely encapsulated in the semantics. We then give an axiom system for program equivalence in the presence of local state that avoids all mention of memory or context and that does not use semantic arguments. The system is complete relative to the underlying flat equational theory. We also indicate briefly how the semantics can be extended to include higher-order functions.

1 Introduction

Much work has been done on the semantics of programs with local state. Most of this work involves intricate storage modeling with pointers and memory cells or complex categorical constructions and the notion of contextual (observable) equivalence. Pitts [10] explains that the formalisms required to determine contextual equivalence are cumbersome because "complete program context" and "observable behavior" are difficult to define formally. This observation is borne out by the complexity of systems for reasoning about program equivalence in the presence of local state.

Seminal work by Meyer and Sieber [8] used the store model of Halpern-Meyer-Trakhtenbrot to prove equivalence of ALGOL procedures with no parameters. Their goal was to formalize informal arguments about the contextual equivalence of programs with block structure.

Building on this work, Mason and Talcott [5, 6, 7] considered a λ-calculus extended with state operations. They formalized context in terms of mem-

ory contexts and variable substitutions that effectively make expressions closed. Two expressions are operationally equivalent if and only if they exhibit this closed form equivalence. By defining axioms in the form of contextual assertions, Mason and Talcott were able to prove the equivalence of several examples of Meyer and Sieber. Some of these contextual assertions correspond to our axioms, however their notion of equivalence still requires the explicit use of context and semantic reasoning.

One key element missing from the work of Meyer and Sieber and Mason and Talcott is the ability to reason about control flow structures such as conditionals and loops. Both models limit themselves specifically to models without such constructs. The axioms presented in [7] reflect this limitation, as they do not have any rules equivalent to our rules for reasoning about iteration.

More recent work of Pitts [9, 10] focuses on the equivalence of ML programs with references using operational semantics. Two major sources of difficulty in proving program equivalence in Pitts' approach are the escape and aliasing of references. Pitts and Stark [11] consider a language that uses local state. Although references are used at a local level, which allows the use of a local invariant to reason about expression equivalence, references are still accessible globally in certain contexts.

Other work focuses on limiting the availability of state to local regions using language constructs and type systems. Aboul-Hosn [2] defines a notion of private state for ML that allows reasoning about equivalence at a functional level instead of a contextual level. The language uses an operational semantics that defines a stack of local states, similar to the way a stack of partial valuations is used as an environment in this paper.

Honsell, Mason, Smith and Talcott [4] mention the difficulty of making assertions about contexts in the presence of the ability to alter references. As a solution, they propose localizing statements about contextual equivalence in the logic. However, in both [2] and [4], this limits the kinds of programs one can reason about.

In this paper, we wish to explore the extent to which *relational semantics* can be used to avoid intricate memory modeling and the explicit use of context in program equivalence proofs. Our objectives are twofold: (i) to define a notion of local variable scoping, along with a purely compositional, fully abstract semantics based on binary relations, such that all contextual considerations are completely encapsulated in the semantics; and (ii) to provide an axiom system for program equivalence in the presence of local state that avoids all mention of memory or context and that does not need to revert to semantic arguments.

Our system is based on schematic Kleene algebra with tests (KAT) augmented with a **let** statement for local variable scoping. The benefits of this approach are that the relational semantics and deductive theory of the standard control-flow constructs reduce to the semantics of the regular and Boolean operators, which are mathematically elegant and have a well-defined and well-studied relational semantics and deductive theory. The only new construct that requires separate treatment is the **let** statement. The usual approach [8, 5, 6, 7] is to reduce the **let** construct to a λ-expression. In the absence of first-class

functions, however, it is much simpler to give a relational semantics that avoids second-order notions. We provide such a definition. The semantics extends the standard relational semantics used in first-order KAT and Dynamic Logic involving valuations of program variables. 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 of the **let** statement.

For most of this paper, we consider the case of first-order programs only. We define the relational semantics of programs with local variable scoping and give a set of simple proof rules that allow **let** statements to be systematically eliminated. Thus the proof system is complete relative to the underlying equational theory without local scoping.

In the presence of higher-order data objects, a similar approach can be taken, but here a state becomes a *closure structure*, a generalization of a stack. However, the semantics can still be defined purely compositionally, without intricate storage structures or explicit mention of context. We briefly explore this alternative 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 *. 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 *scoping expressions* in the inductive definition of programs. A *scoping expression* is an expression

$$\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end} \tag{1}$$

where p is a program, the x_i are distinct program variables, and the t_i are terms, $1 \le i \le n$.

Operationally, when entering the scope (1), 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. 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 $f : \mathsf{Var} \to |\mathfrak{A}|$,

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

For an environment σ,

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

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

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

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

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

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

$$\begin{aligned} &[\![\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}]\!] \\ &\quad = \quad \{(\sigma, \mathsf{tail}(\tau)) \mid \sigma(t_i) \text{ is defined, } 1 \leq i \leq n, \text{ and } (f :: \sigma, \tau) \in [\![p]\!]\}, \end{aligned} \tag{2}$$

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

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 $R(t_1, \ldots, t_n)$,

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

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

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

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

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

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

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

Example 2.1 Consider the program

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

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

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

□

Lemma 2.2 *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. □

3 Axioms and Basic Properties

In this section we present a simple, purely syntactic 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.

Axiom System 3.1

(i) If the y_i are distinct and do not occur in p, $1 \le i \le n$, then the following two programs are equivalent:

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

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

(ii) If y does not occur in s, then the following two programs are equivalent:

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

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

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

(iii) If x does not occur in s, then the following two programs are equivalent:

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

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

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

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

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

(vi) If x does not occur in pr, then the following two programs are equivalent:

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

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

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

(viii) 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.

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

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

(x) 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 3.2 *Axioms 3.1 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 (viii), 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

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

the last by a straightforward infinitary generalization of (vii). 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 (v). Now suppose it is true for n. Then

$$\begin{aligned}&(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^{n+1} \\ &= (\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^n; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \\ &= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \quad (3) \\ &= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^n; x := t; p\ \mathsf{end} \quad (4) \\ &= \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^{n+1}\ \mathsf{end}\end{aligned}$$

where (3) follows from the induction hypothesis and (4) follows from the identity

$$\mathsf{let}\ x = a\ \mathsf{in}\ q\ \mathsf{end}; \mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} = \mathsf{let}\ x = a\ \mathsf{in}\ q; x := t; p\ \mathsf{end} \quad (5)$$

To justify (5), 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. □

Lemma 3.3

(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:*

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

4 Flattening (Globalization)

To prove equivalence of two programs p, q with scoping, we take the following approach. 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 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 3.1 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 (vi)–(viii). Adjacent scoping expressions can be combined using (iii) and (iv). Finally, all bindings can be put into the form $x{=}a$ using (ix).

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 4.1 *Axioms 3.1 are sound and complete for program equivalence relative to the underlying equational theory without local scoping.*

5 Higher-Order Functions

One can extend these ideas to provide a natural relational semantics for a programming language with first-class functions. In this section we describe briefly how such a relational semantics might look.

This treatment contrasts sharply with other contemporary functional or denotational approaches (see for example [12, 13, 3]). One distinguishing aspect of our approach is that functions and data are not conflated; we distinguish between expressions that can denote values and those that can denote programs. This allows us to give a development that aligns more closely with the

procedural view of computation (computation as state manipulation) without abandoning the functional view (computation as evaluation). This is useful even for languages such as ML that are nominally functional. This approach will be beneficial if it allows a more axiomatic treatment of program equivalence or a treatment of partial correctness involving Hoare-style pre- and postconditions, which would currently be difficult or impossible with the conventional purely functional approach.

As above, let $\mathfrak{A}$ be a first-order domain of computation of some fixed signature Σ consisting of constants, function symbols, and relation symbols of various types. For notational simplicity in this development, we assume that all function and relation symbols are unary.

Syntactically, there are two classes of expressions: *value expressions* and *program expressions*. Semantically, value expressions will denote binary relations between execution states and values, and program expressions will denote binary relations between execution states and execution states. Intuitively, a value expression can be evaluated in an execution state, and the binary relation gives the set of possible values. It is a set because of nondeterminism in the language. Similarly, the binary relation associated with a program expression gives for each input state the set of possible output states.

5.1 Syntax

Value types are either *base types* as determined by the signature of $\mathfrak{A}$ or *procedural types* of the form $\sigma \to \tau$ or $\sigma \to \mathsf{void}$, where σ and τ are value types. We assume the existence of infinitely many variables of all value types.

An atomic value expression of procedural type is either (i) a variable of procedural type, (ii) a function symbol f of $\mathfrak{A}$, or (iii) a λ-term of the form $\lambda x.p$ or $\lambda x.p; e$, where p is a program expression and e is a value expression. In (iii), the former form is for methods with no return value (or return value void) and the the latter is for methods with return value e. Note that these are value expressions, *not* program expressions.

Formally, value expressions and program expressions are defined by mutual induction as follows. A *value expression* is either

(i) a variable,

(ii) a constant or function symbol of $\mathfrak{A}$,

(iii) a λ-term $\lambda x.p$, where x is a variable and p is a program expression,

(iv) a λ-term $\lambda x.p; e$, where x is a variable, p is a program expression, and e is a value expression,

(v) an application $P(d)$, where P is a procedural expression with non-void return type and d is a value expression of the appropriate type for P.

A *program expression* is either

(i) an assignment $x := d$, where x is variable and d is a value expression of the same type,

(ii) a test $R(d)$, where R is a relation symbol of the signature of $\mathfrak{A}$ and d is a value expression of the appropriate type for R,

(iii) a nondeterministic choice $p + q$, where p and q are program expressions,

(iv) a sequential composition $p\,;\,q$, where p and q are program expressions,

(v) an iteration p^*, where p is a program expression,

(vi) an application $P(d)$, where P is a procedural expression with either a void or non-void return type and d is a value expression of the appropriate type for P.

As mentioned, $\lambda x.p$ and $\lambda x.p; e$ are only value expressions, not program expressions. The application $(\lambda x.p)(d)$ is only a program expression, but the application $(\lambda x.p; e)(d)$ is both a program expression and a value expression.

In the presence of higher-order functions, we can omit primitive let expressions, since they are redundant by a standard encoding:

$$\begin{aligned} \mathsf{let}\ x = d\ \mathsf{in}\ p\ \mathsf{end} &= (\lambda x.p)(d) \\ \mathsf{let}\ x = d\ \mathsf{in}\ p; e\ \mathsf{end} &= (\lambda x.p; e)(d). \end{aligned}$$

5.2 Closure Structures

Before we can define the binary relation semantics of our language, we need to generalize the notion of execution state from a stack of finite valuations as in Section 2 to a certain kind of pointed tree called a *closure structure*. This definition is directly motivated by the operational semantics of ML, Scheme, and other languages with static binding, in which the environment of a method declaration is saved with the compiled method for the purpose of evaluating free variables when the method is called. It is similar in many respects to the untyped operational semantics of Abadi and Cardelli [1, Ch. 10]. All finite valuations in the tree will have singleton domains because of our simplifying assumption that functions are unary, but this restriction is inessential.

Formally, a *closure structure* is a pair $\sigma = (T, \alpha)$, where T is a tree of bindings of the form $x = c$, where x is a variable and c is a value of the same type, and α is a pointer into T. The pointer α is called the *active pointer* of σ and is denoted $\mathsf{active}(\sigma)$. The values c occurring in bindings in T are either elements and functions of $\mathfrak{A}$ or pairs $(\lambda x.p,\ \beta)$ or $(\lambda x.p; e,\ \beta)$, where β is a pointer into T. The root of any tree T is nil. The empty closure structure is $(\mathsf{nil},\ \alpha)$ where α is a pointer to the root nil.

Every pointer α into T determines an environment as defined in Section 2, namely the environment consisting of the bindings along the path from α to the root of T. The environment of σ that $\mathsf{active}(\sigma)$ points to is called the *active environment* of σ. The pointer β in a value $(\lambda x.p,\ \beta)$ or $(\lambda x.p; e,\ \beta)$ is included in

order to recall the environment in which the expression was evaluated. Although this environment may change over the lifetime of the binding due to variable assignments, the pointer does not.

We define the following operations on closure structures. If f is a binding and σ is a closure structure, $f :: \sigma$ is the closure structure obtained by appending f to the active environment of σ and updating $\mathsf{active}(\sigma)$ to point to f. The inverse operations are the projections $\mathsf{head}(f :: \sigma) = f$, $\mathsf{tail}(f :: \sigma) = \sigma$. Variable lookup and rebinding are defined inductively exactly as in Section 2, using the active environment. The value of variable x in closure structure σ is denoted $\sigma(x)$, and the result of rebinding x in σ to the new value c is denoted $\sigma[x/c]$. As in Section 2, if there is no binding of x in the active environment, then $\sigma(x)$ is undefined and rebinding has no effect.

5.3 Semantics

Let CS denote the set of closure structures. Each value expression e denotes a binary relation $[\![e]\!]$ relating closure structures and values, and each program expression p denotes a binary relation $[\![\![p]\!]\!]$ relating (input) closure structures and (output) closure structures. The definitions are mutually inductive.

For value expressions:

(i) If x is a variable, $[\![x]\!] = \{(\sigma, \sigma(x)) \mid \sigma \in \mathsf{CS},\ \sigma(x) \text{ is defined}\}$.

(ii) If f is a constant or function symbol of $\mathfrak{A}$, $[\![f]\!] = \{(\sigma, f^{\mathfrak{A}}) \mid \sigma \in \mathsf{CS}\}$.

(iii) $[\![\lambda x.p]\!] = \{(\sigma, (\lambda x.p, \mathsf{active}(\sigma))) \mid \sigma \in \mathsf{CS}\}$.

(iv) $[\![\lambda x.p; e]\!] = \{(\sigma, (\lambda x.p; e, \mathsf{active}(\sigma))) \mid \sigma \in \mathsf{CS}\}$.

(v) If f is a function symbol of $\mathfrak{A}$ and d is a value expression of the appropriate type for f, then $[\![f(d)]\!] = \{(\sigma, f^{\mathfrak{A}}(c)) \mid (\sigma, c) \in [\![d]\!]\}$.

(vi) If P is a procedural expression with non-void return type and d is a value expression of the appropriate type for P, then

$$\begin{aligned}[\![P(d)]\!] \;=\; &\{((T,\alpha), b) \mid ((T,\alpha), c) \in [\![d]\!],\\ &\qquad ((T,\alpha), (\lambda x.p; e,\ \beta)) \in [\![P]\!],\\ &\qquad ((x = c) :: (T,\beta),\ b) \in [\![\![p]\!]\!] \circ [\![e]\!]\}\\ &\cup\ \{((T,\alpha), f(c)) \mid ((T,\alpha), c) \in [\![d]\!],\ ((T,\alpha), f) \in [\![P]\!]\}.\end{aligned}$$

For program expressions:

(i) $[\![\![x := d]\!]\!] = \{(\sigma, \sigma[x/a]) \mid (\sigma, a) \in [\![d]\!],\ \sigma(x) \text{ is defined}\}$.

(ii) $[\![\![R(d)]\!]\!] = \{(\sigma, \sigma) \mid (\sigma, a) \in [\![d]\!] \text{ and } R^{\mathfrak{A}}(a)\}$.

(iii) $[\![\![p + q]\!]\!] = [\![\![p]\!]\!] \cup [\![\![q]\!]\!]$.

(iv) $[\![\![p\,;\, q]\!]\!] = [\![\![p]\!]\!] \circ [\![\![q]\!]\!]$.

(v) $[\![p^*]\!] = \bigcup_n [\![p]\!]^n$.

(vi) If P is a procedural expression with void return type and d is a value expression of the appropriate type for P, then

$$[\![P(d)]\!] = \{((T,\alpha),(S,\alpha)) \mid ((T,\alpha), c) \in [\![d]\!], \; ((T,\alpha), (\lambda x.p, \beta)) \in [\![P]\!], \; ((x = c) :: (T,\beta), (x = d) :: (S,\beta)) \in [\![p]\!]\}.$$

The definition is the same for P a procedural expression with non-void return type; the return expression in the λ-term is ignored.

For example, let P be a variable of procedural type with void return value, and let d be a value expression of the appropriate input type for P. The definition of $[\![P(d)]\!]$ captures the following operational intuition. Given an initial execution state described by a closure structure $\sigma = (T, \alpha)$, the halting states are all states (S, α) obtained as follows. First, we evaluate d in the state σ to obtain a value c. The set of all such c we might obtain are all those such that $(\sigma, c) \in [\![d]\!]$. Then we evaluate P in the state σ to obtain a value, say $(\lambda x.p, \beta)$, consisting of a procedural expression $\lambda x.p$ and a pointer β to the active environment at the time the value $(\lambda x.p, \beta)$ was created. For instance, we might have previously executed an assignment $P := \lambda x.p$, in which case β would be the active pointer at the time of the assignment. We then change the active environment to β to get the closure structure (T, β), bind the formal parameter x to the argument c and append this binding to the closure structure to get $(x = c) :: (T, \beta)$, then run p until it halts, yielding an output state $(x = d) :: (S, \beta)$. The set of all possible output states is the set of all $(x = d) :: (S, \beta)$ such that $((x = c) :: (T, \beta), (x = d) :: (S, \beta)) \in [\![p]\!]$. As in Lemma 2.2, one can prove inductively that the shape of the closure structure does not change, only the bindings. The binding $x = d$ is then popped and the active pointer restored to α.

A *context* $C[\text{-}]$ is just a program expression with a distinguished free program variable (note that no other program variables exist in the language—all other variables are value variables). The following result asserts that the semantics is fully abstract, although full abstraction is not really the interesting issue here, but rather that contexts are superfluous in program equivalence arguments.

Theorem 5.1 *For program expressions p and q, $[\![C[p]]\!] = [\![C[q]]\!]$ for all contexts $C[\text{-}]$ iff $[\![p]\!] = [\![q]\!]$.*

Proof. The proof of this theorem is actually quite straightforward. The direction ($\rightarrow$) is trivial by taking $C[\text{-}]$ to be the null context. 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. □

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Kamal Aboul-Hosn. Programming with Private State. Honors Thesis, The Pennsylvania State University, December 2001.

[3] Marcelo P. Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proc. 14th Symp. Logic in Computer Science (LICS'99)*, pages 193–202, July 1999.

[4] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

[5] 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.

[6] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Jon G. Riecke and Anders Sandholm. A relational account of call-by-value sequentiality. In *Proc. 12th Symp. Logic in Computer Science (LICS'97)*, pages 258–267, June 1997.

[13] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Proc. 22th Symp. Principles of Programming Languages (POPL'95)*, pages 1–12, January 1995.