Relational Semantics for Higher-Order Programs

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 higher-order 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 show how a relatively simple relational semantics can be used to avoid these complications. We provide a natural relational semantics for a programming language with higher-order functions. The semantics is purely compositional, with all contextual considerations completely encapsulated in the state. We show several equivalence proofs using this semantics based on examples of Meyer and Sieber (1988).

1 Introduction

Reasoning about higher-order programs with local state is an important and difficult problem that has garnered 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]).

Work in modeling local state dates back over thirty years. Early seminal work by Meyer and Sieber [12] 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. One of the most important contributions of their work was the introduction of seven examples that exemplify the subtleties in reasoning about programs with local state. These classical examples have become the preferred standard against which to evaluate models that address the problem.

Much early attention focused on the use of denotational semantics to model a set of storage locations [4–7]. The inability to prove some simple program

equivalences using traditional denotational techniques led several researchers to take a categorical approach [8–10]. See [11] 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 of several examples of Meyer and Sieber. Pitts and Stark [1, 16–18] also use operational semantics.

Others have used 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 and higher-order constructs.

In this paper, we wish to explore the extent to which *relational semantics* can be used to avoid intricate memory modeling, category theory, and the explicit use of context in program equivalence proofs. Relational semantics combine the expressiveness of denotational semantics with the more intuitive understanding of operational semantics. Our objective is to 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 provide a natural relational semantics for a programming language with higher-order functions in Section 3. This treatment contrasts sharply with other contemporary functional or denotational approaches (see for example [23–25]). 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. Our semantics allows destructive updates, but no aliasing.

Fully compositional relational semantics have been quite popular for first-order imperative programs (see for example [26] and references therein), but to our knowledge this is the first attempt to provide semantics in this style to higher-order programs.

We are ultimately interested in moving toward a more axiomatic treatment of program equivalence and partial correctness for higher-order programs in the style of Hoare logic or Kleene algebra with tests [27]. Our compositional program operators are based on the Kleene algebra operators (see [27, 26]), which have well-understood relational models and are simpler and more amenable to axiomatic treatment than conventional programming constructs. We take some initial steps in this direction in Section 4, in which we prove six of Meyer and Sieber's seven examples using relational semantics. The only example we cannot handle is the one involving aliasing, since our semantics does not treat aliasing at present.

2 Syntax

2.1 Types

A *type* is either a *base type* denoting an individual element of the domain of computation or a *functional type* of the form $\mathsf{s} \to \mathsf{t}$, where s and t are types or void. The notation void is to accommodate methods with no arguments and/or no return value, but it is not itself a type. We assume the existence of infinitely many variables of each type.

Expressions are either *value expressions* or *program expressions*. These two sets of expressions are disjoint and are defined by mutual induction.

2.2 Value Expressions

Value expressions must be well-typed. Let Σ be a first-order signature consisting of a collection of function, relation, and constant symbols. A *value expression* is either

(i) a variable,
(ii) a symbol of the signature Σ,
(iii) a λ-term of the form $\lambda x.p$, $\lambda x.p; e$, $\lambda().p$, or $\lambda().p; e$, where x is a variable, p is a program expression, and e is a value expression,
(iv) an application $P(d)$, where P is a value expression of functional type with non-void return type and d is a value expression of the appropriate input type for P,
(v) an application $P()$, where P is a value expression of functional type with non-void return type and void input.

Evaluation of an expression of the form (i)–(iii) is immediate and without side effects. In (iii), the forms $\lambda x.p$ and $\lambda().p$ are for methods with no return value (or return value void) and the forms $\lambda x.p; e$ and $\lambda().p; e$ are for methods with return value e. The forms $\lambda().p$ and $\lambda().p; e$ are parameterless methods. In general, the process of evaluating a value expression (iv) or (v) can have side effects, which manifest themselves as a change of state.

2.3 Program Expressions

Program expressions differ from value expressions in that they do not yield a value. However, their execution generally results in a change of state.

Syntactically, 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 Σ and d is a value expression of the appropriate input 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 functional expression with void return type and d is a value expression of the appropriate input type for P,
(vii) an application $P()$, where P is a functional expression with void input and return type.

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

In the presence of higher-order functions, we can encode let expressions by a standard encoding:

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

3 Relational Semantics

The *domain of computation* is a first-order structure $\mathfrak{A}$ of signature Σ. Each symbol of Σ is interpreted as a function, relation, or constant of $\mathfrak{A}$ of the appropriate type.

3.1 Closure Structures

Before we can give the semantics, we must define what we mean by a state of execution. Informally, a state is a structure that contains all the variable/value bindings that have been created up to that point, along with specific rules for lookup, new binding creation, and destructive update. We will call these *closure structures*. Programs will be interpreted as relations on closure structures. The 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; see for example [28, Ch. 10].

Formally, a *closure structure* is a triple $\sigma = (T, \alpha, s)$, where T is a tree, α is a reference to a node in T, and s is a stack of references to nodes in T.

Each node of T (except the root) is an object containing

- a binding of the form $x = c$, where x is a variable and c is a value of the same type, and
- a reference to its parent in T.

Distinct nodes are different objects, but may represent the same binding and may have the same parent. We use $\alpha, \beta, \ldots$ to refer to nodes of T and $\sigma, \tau, \ldots$ to refer to closure structures.

Every node α of T uniquely determines an *environment*, which is the list consisting of α and all its ancestors back to the root of T. We denote this environment also by α. This slight abuse of notation should cause no confusion, since

there is a one-to-one correspondence between nodes in T and the environments they determine.

It is important to note that we have not defined an environment as a list of bindings. As distinct nodes can represent the same binding, so can distinct environments represent the same list of bindings.

The root of T, denoted ε, represents the empty environment with no bindings. It is the terminal node of all environments in T.

The empty closure structure is $(\varepsilon, \varepsilon, \texttt{[]})$, where ε is the root and `[]` is the empty stack.

The environment α in a closure structure $\sigma = (T, \alpha, s)$ is called the *active environment* of σ and is denoted $\mathsf{actv}(\sigma)$. In Section 3.4 below, we will describe the operations of lookup and rebinding on closure structures. These operations are always performed in the active environment.

The set of closure structures is denoted CS.

3.2 Values

The values c occurring in bindings are either

(i) elements and functions of the domain of computation $\mathfrak{A}$, or
(ii) pairs (t, β), where t is a λ-expression of the form $\lambda x.p$, $\lambda x.p; e$, $\lambda().p$, or $\lambda().p; e$ and β is a reference to a node in T.

Values of class (i) are called *intrinsic values*, and those of class (ii) are called *closures*.

A closure (t, β) is created when the expression t is evaluated. The reference β is included in order to recall the environment that was active at the time of the evaluation. That environment will be used in future calls to interpret the free variables of t. Although the bindings in this environment may change over the lifetime of the object due to variable assignments, the reference β does not.

Symbols $f, g, \ldots$ range over intrinsic functions. Since we have postulated Σ as a first-order signature, closures, which are of functional type, cannot be arguments of intrinsic functions. All higher-order functions must be constructed using λ-expressions.

The set of values is denoted Val.

3.3 Accessibility

A node of a closure structure is *accessible* if it is reachable starting from the active environment or from a reference on the stack and following parent references or references β in closures (t, β). Note that any descendant of an inaccessible node is inaccessible. Two closure structures are considered equivalent if their accessible substructures are isomorphic; that is, if there is a one-to-one correspondence between accessible nodes of their trees and between their stack entries and active environments that preserves stack order and all reference relationships and binding values (environment references in closures are mapped appropriately under the isomorphism). Equivalence modulo accessibility can be viewed as a kind

of mathematical garbage collection, although we do not postulate any explicit mechanism for garbage collection.

The purpose of the stack is to ensure the persistence of nodes across computations in which those nodes might otherwise become inaccessible. We will give a more precise explanation when we give the relational semantics below.

3.4 Operations on Closure Structures

Our relational semantics is defined in terms of the following low-level operations on closure structures.

If x is variable, c is a value, and α is an environment in σ, then $x = c : \alpha$ denotes the environment obtained by creating a new node with binding $x = c$ and prepending it to α. Whenever this occurs, a reference to α is available on top of the stack of σ, along with a reference to γ in the case c is a closure (t, γ). These references are popped (or just the reference to α, if c is an intrinsic value) and a reference to the newly created node is pushed onto the stack.

If σ is a closure structure and β is an environment in σ, then $\beta + \sigma$ denotes the result of popping β off the stack (it will always be there when this operation is applied), pushing the current active environment on the stack, and making β the new active environment. Thus we can think of this operation simply as switching the active environment with the environment on top of the stack.

These two operations are most commonly used in tandem to create a new binding. In this case, $(x = c : \alpha) + \sigma$ denotes the closure structure obtained from σ by creating a new node with binding $x = c$, prepending this node to α, then making this the new active environment. Before this operation, references to α and γ if c is a closure (t, γ) are available on top of the stack. In the special case in which $\alpha = \mathsf{actv}(\sigma)$, we abbreviate this by $(x = c) + \sigma$. The cumulative effect on the stack is to pop one or two elements, depending on whether c was an intrinsic value or a closure, respectively, and pushing the old active environment.

All evaluation of and assignment to variables is done in $\mathsf{actv}(\sigma)$, the active environment of σ. When evaluating a variable x, the value is the one bound to the first (most recently bound) occurrence of x in $\mathsf{actv}(\sigma)$. This value is denoted $\sigma(x)$. If x is not bound in $\mathsf{actv}(\sigma)$, then $\sigma(x)$ is undefined. When assigning to a variable x, we destructively rebind the first occurrence of x in $\mathsf{actv}(\sigma)$ to its new value. It is important to note that this is done destructively, not functionally: the list of nodes in $\mathsf{actv}(\sigma)$ is not changed, but only the value in the binding of one of the nodes. We denote the result of rebinding x to the new value a in closure structure σ by $\sigma[x/a]$. In addition, if a is a closure (t, β), then the stack is popped; in this case the top element will always be β. If x is not bound in $\mathsf{actv}(\sigma)$, the rebinding operator $[x/a]$ has no effect.

In real life, any attempt to evaluate or assign to an undefined variable (one that is not in the domain of the active environment) would result in a runtime error. The relational semantics to be given below will ensure that there will be no tuple in the relation corresponding to the program with that input state.

The value of a term t in the language of $\mathfrak{A}$ in a closure structure σ is denoted $\sigma(t)$ and is defined by structural induction on t in the usual way. Note that $\sigma(t)$ is defined iff x is bound in $\mathsf{actv}(\sigma)$ for all variables x occurring in t.

The operation $\mathsf{rest}(\sigma)$ just restores an earlier active environment by popping the stack and setting the active environment to that value. The current active environment is discarded. Curiously, $\mathsf{rest}(\beta + \sigma)$ is not necessarily equivalent to σ, since β may no longer be accessible.

We give a skeleton implementation in the appendix for illustrative purposes. Equivalence of closure structures modulo accessibility could be implemented by a deep equality test, although care must be taken due to circularities that can be introduced by destructive updates.

3.5 Semantics

Let CS denote the set of closure structures and Val the set of values. Each value expression e denotes a binary relation

$$[\![e]\!] \subseteq \mathsf{CS} \times (\mathsf{CS} \times \mathsf{Val}) \tag{1}$$

relating input states with (output state, value) pairs. We write $(\sigma, (\tau, c))$ simply as (σ, τ, c). Each program expression p denotes a binary relation

$$[\![\![p]\!]\!] \subseteq \mathsf{CS} \times \mathsf{CS} \tag{2}$$

relating input states with output states. The definitions are mutually inductive. Value expressions e also denote binary relations of the form (2), but these are derived immediately from (1) by projecting out the value:

$$[\![\![e]\!]\!] = \{(\sigma, \tau') \mid (\sigma, \tau, c) \in [\![e]\!]\},$$

where $\tau' = \tau$ if c is an intrinsic value, and is τ with the stack popped if c is a closure (t, β). In the latter case, the value that is popped will always be β.

3.6 Value Expressions

(i) If x is a variable, $[\![x]\!] = \{(\sigma, \sigma', \sigma(x)) \mid \sigma \in \mathsf{CS},\ \sigma(x) \text{ is defined}\}$, where $\sigma' = \sigma$ if $\sigma(x)$ is an intrinsic value, or σ with β pushed on the stack if $\sigma(x)$ is a closure (t, β).

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

(iii) If t is a λ-expression of the form $\lambda x.p$, $\lambda x.p; e$, $\lambda().p$, or $\lambda().p; e$, then

$$[\![t]\!] = \{(\sigma, \sigma', (t, \mathsf{actv}(\sigma))) \mid \sigma \in \mathsf{CS}\},$$

where σ' is σ with $\mathsf{actv}(\sigma)$ pushed onto the stack.

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

$$\begin{aligned}[\![P(d)]\!] = \{(\sigma, \mathsf{rest}(\tau), b) \mid\ & \exists\rho\ \exists\upsilon\ \exists c\ \exists(\lambda x.p; e, \beta) \\ & (\sigma, \rho, (\lambda x.p; e, \beta)) \in [\![P]\!],\ (\rho, \upsilon, c) \in [\![d]\!], \\ & ((x = c : \beta) + \upsilon, \tau, b) \in [\![\![p]\!]\!] \circ [\![e]\!]\} \\ \cup\ \{(\sigma, \tau, f(c)) \mid\ & \exists\rho\ (\sigma, \rho, f) \in [\![P]\!],\ (\rho, \tau, c) \in [\![d]\!]\}.\end{aligned}$$

(v) If P is a functional expression with non-void return type and no parameter, then

$$\begin{aligned}\llbracket P() \rrbracket = \{(\sigma, \mathsf{rest}(\tau), b) \mid\ & \exists \rho\ \exists (\lambda().p; e, \beta) \\ & (\sigma, \rho, (\lambda().p; e, \beta)) \in \llbracket P \rrbracket,\ (\beta + \rho, \tau, b) \in \llbracket p \rrbracket \circ \llbracket e \rrbracket\} \\ \cup\ \{(\sigma, \tau, f()) \mid (\sigma, \tau, f) \in \llbracket P \rrbracket\}.\end{aligned}$$

In (iv) and (v), the composition operator in the expression $\llbracket p \rrbracket \circ \llbracket e \rrbracket$ is ordinary binary relation composition; recall that $\llbracket e \rrbracket$ is officially a binary relation. Thus

$$\llbracket p \rrbracket \circ \llbracket e \rrbracket = \{(\sigma, \tau, c) \mid \exists \rho\ (\sigma, \rho) \in \llbracket p \rrbracket,\ (\rho, \tau, c) \in \llbracket e \rrbracket\}.$$

The definition of $\llbracket P(d) \rrbracket$ in (iv) captures the following operational intuition. Given an initial execution state described by a closure structure σ, the halting states and output values are all those obtained as follows. First, we evaluate P in the state σ to obtain a value, say $(\lambda x.p; e,\ \beta)$, consisting of a value expression $\lambda x.p; e$ and a reference β to the active environment at the time the value $(\lambda x.p; e,\ \beta)$ was created. For instance, if P is a variable, we might previously have executed an assignment $P := \lambda x.p; e$, where β was the active environment at the time of the assignment. We also obtain a new state ρ. In general the new state may be different, since the evaluation of P might have had side effects. There may be several possible values and states obtained in this way due to nondeterminism in the evaluation of P, but the set of all such values and states we might obtain are given by all elements of $\llbracket P \rrbracket$ with first component σ.

Then we evaluate the argument expression d in the resulting state ρ to obtain a value c and an output state υ. The stack is used to preserve β across this computation. We then create a new node with binding $x = c$, where x is the formal parameter and c is the argument value just computed, and prepend this binding to the environment β to obtain the environment $x = c : \beta$. This becomes the new active environment, and the state is now $(x = c : \beta) + \upsilon$. We run $p; e$ starting in this state until it halts, yielding an output state τ and value b. The stack is then popped to restore the previous active environment, giving $\mathsf{rest}(\tau)$, and this is the final output state.

3.7 Program Expressions

(i) $\llbracket x := d \rrbracket = \{(\sigma, \tau[x/a]) \mid (\sigma, \tau, a) \in \llbracket d \rrbracket,\ \sigma(x) \text{ is defined}\}$. Recall that if a is a closure, then the stack of τ is popped in the formation of $\tau[x/a]$.
(ii) $\llbracket R(d) \rrbracket = \{(\sigma, \tau) \mid (\sigma, \tau, a) \in \llbracket d \rrbracket,\ R^{\mathfrak{A}}(a)\}$.
(iii) $\llbracket p + q \rrbracket = \llbracket p \rrbracket \cup \llbracket q \rrbracket$.
(iv) $\llbracket p\ ;\ q \rrbracket = \llbracket p \rrbracket \circ \llbracket q \rrbracket$.
(v) $\llbracket p^* \rrbracket = \bigcup_{n \geq 0} \llbracket p \rrbracket^n =$ the reflexive transitive closure of $\llbracket p \rrbracket$.
(vi) If P is a functional expression with void return type and d is a value expression of the appropriate input type for P, then

$$\begin{aligned}\llbracket P(d) \rrbracket = \{(\sigma, \mathsf{rest}(\tau)) \mid\ & \exists \rho\ \exists \upsilon\ \exists c\ \exists (\lambda x.p, \beta) \\ & (\sigma, \rho, (\lambda x.p, \beta)) \in \llbracket P \rrbracket,\ (\rho, \upsilon, c) \in \llbracket d \rrbracket, \\ & ((x = c : \beta) + \upsilon, \tau) \in \llbracket p \rrbracket\} \\ \cup\ \{(\sigma, \tau) \mid \exists \rho\ \exists f\ & (\sigma, \rho, f) \in \llbracket P \rrbracket,\ (\rho, \tau) \in \llbracket d \rrbracket\}.\end{aligned}$$

(vii) If P is a functional expression with void return type and no parameter, then

$$\begin{aligned}[[P()]] = \{(\sigma, \mathsf{rest}(\tau)) \mid\ & \exists \rho\ \exists (\lambda().p, \beta) \\ & (\sigma, \rho, (\lambda().p, \beta)) \in [P],\ (\beta + \rho, \tau) \in [[p]]\} \\ \cup\ & \{(\sigma, \tau) \mid \exists f\ (\sigma, \tau, f) \in [P]\}.\end{aligned}$$

The Kleene algebra operators $+, ;, ^*$ have been used here for mathematical simplicity. It is well known how to define more conventional programming constructs such as conditional branches and while loops from them; see for example [27, 26].

3.8 Discussion

The shape of the tree can change during a computation, as new nodes can be added or previously accessible nodes can become inaccessible. This is the reason we must consider equivalence modulo accessibility. However, there are strong invariants on the active environment and the stack:

- For $[[p]]$, both the active environment and the stack are preserved from input to output.
- For $[p]$, the active environment is preserved from input to output. The stack is also preserved if the output value is intrinsic. Otherwise, if the output value is a closure (t, β), then the output stack consists of the input stack with a reference to β pushed on top.

These can be verified by induction on the structure of p.

The stack is needed to preserve active environments across function calls. It is also needed to preserve β across the evaluation of the argument d in 3.6(iv) and 3.7(vi) when the function to be applied is a closure (t, β).

One might well ask: In the preservation of β across calls, why is it not necessary to preserve t as well? This is certainly a legitimate question. The answer is that it *would* be necessary in any real implementation. However, here we are only trying to define a binary input/output relation, and the mathematical definitions 3.6(iv) and 3.7(vi) do this adequately without any explicit mechanism in closure structures for remembering t.

So why then does the same argument not apply to β? In an earlier version of this work, we thought that it did. However, there is a subtlety related to our assumption regarding equivalence modulo accessibility. We must ensure that in any triple $(\sigma, \rho, (t, \beta)) \in [P]$, the node β is accessible in ρ and remains accessible throughout the calculation $(\rho, \upsilon, c) \in [d]$. Otherwise, the subsequent operation $(x = c : \beta) + \upsilon$ would not make sense, since the formalism does not keep track of the correspondence between nodes of ρ and those of υ. The value expression $\mathsf{let}\ x = 0\ \mathsf{in}\ \lambda().x\ \mathsf{end}$ provides an example of a P for which this is an issue. The corresponding closure contains a reference to the binding $x = 0$, but this node would be inaccessible after the evaluation of the expression if not for the stack.

3.9 Eliminating Context

The relational semantics presented in Sections 3.6 and 3.7 captures all contextual information in the state, allowing us to reason about programs with local state in a purely compositional way without considering their context. Formally, a *context* $C[\text{-}]$ is just a program or value expression with a distinguished free program variable. It is easy to see that for any program expressions p and q, $[\![C[p]]\!] = [\![C[q]]\!]$ for all contexts $C[\text{-}]$ iff $[\![p]\!] = [\![q]\!]$. For the direction ($\Rightarrow$), take $C[\text{-}]$ to be the trivial context consisting of a single program variable. The converse 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.

3.10 An Example

Consider the program

$$\begin{array}{ll} \mathsf{let} & y = 4 \\ & f = \lambda z.(y := y + z\,;\; x := y) \\ \mathsf{in} & f(1)\,;\; x \\ \mathsf{end} & \end{array} \tag{3}$$

where x, y, z, and f are all distinct variables. Translating this program into a λ-expression, we obtain

$$\lambda y.(\lambda f.(f(1)\,;\; x)\;\; \lambda z.(y := y + z\,;\; x := y))\;\; (4).$$

First we give an operational account of the computation. Suppose the input state is σ with active environment α. The expression is an application of a function of type $\mathsf{int} \to \mathsf{int}$, thus 3.6(iv) applies. We first evaluate the outermost λ-expression t, which according to 3.6(iii) yields the value (t, α). Then the argument 4 is evaluated, giving value 4. The formal parameter y is bound to the argument 4 and prepended to the environment α in the closure, giving a new active environment $y = 4 : \alpha$, which we call β. The old active environment α is saved on the stack.

Next, we look at the body of the λ-expression t, namely

$$\lambda f.(f(1)\,;\; x)\;\; \lambda z.(y := y + z\,;\; x := y).$$

This is another application, but in this case, the argument is itself a function. We prepend the binding $f = (\lambda z.(y := y + z\,;\; x := y),\ \beta)$ to the active environment β to get a new active environment γ.

The semantics of the body of the function we are applying is the composition $[\![f(1)]\!] \circ [\![x]\!]$. For $f(1)$, we look up f in the active environment γ, retrieve its value $(\lambda z.(y := y + z\,;\; x := y),\ \beta)$, prepend the binding $z = 1$ to β to get the environment δ, then evaluate the body in the environment δ. Note that y and z are bound in δ but not f (unless f was bound in the original active

environment of σ). Now $[\![y := y + z\,;\; x := y]\!]$ will rebind x and y in δ to the value 5, provided x was bound in the original active environment of σ. If not, then there is no output state corresponding to σ. Let τ be the resulting state. The active environment of τ is δ, so $\tau(x) = \tau(y) = 5$.

Now x has value semantics $[\![x]\!] = \{(\sigma, \sigma, \sigma(x)) \mid \sigma \in \mathsf{CS}\}$. One of these tuples is $(\tau, \tau, 5)$. Composing with $[\![f(1)]\!]$, we get an output state τ and corresponding value $\tau(x) = 5$. The stack is popped twice, yielding $\mathsf{rest}(\mathsf{rest}(\tau)) = \sigma[x/5]$ after garbage-collecting the inaccessible bindings of f and y. The value semantics of the entire program contains the tuple $(\sigma, \sigma[x/5], 5)$.

Now we do the same thing calculationally, using the algebraic properties of relations and properties of closure structures. Substituting

$$\lambda y.(\lambda f.(f(1)\,;\; x)\;\;\lambda z.(y := y + z\,;\; x := y))$$

for P and 4 for d in 3.6(iv) and simplifying, we obtain

$$\begin{aligned}
&[\![\lambda y.(\lambda f.(f(1)\,;\; x)\;\;\lambda z.(y := y + z\,;\; x := y))\;\;(4)]\!] \\
&= \{(\sigma, \mathsf{rest}(\tau), b) \mid ((y = 4) + \sigma, \tau, b) \in \\
&\qquad\qquad [\![\lambda f.(f(1)\,;\; x)\;\;\lambda z.(y := y + z\,;\; x := y)]\!]\}.
\end{aligned} \tag{4}$$

Using the same rule with $\lambda f.(f(1)\,;\; x)$ for P and $\lambda z.(y := y + z\,;\; x := y)$ for d, we obtain

$$\begin{aligned}
&[\![\lambda f.(f(1)\,;\; x)\;\;\lambda z.(y := y + z\,;\; x := y)]\!] \\
&= \{(\theta, \mathsf{rest}(\eta), b) \mid (f = (\lambda z.(y := y + z\,;\; x := y), \mathsf{actv}(\theta))) + \theta,\ \eta,\ b) \\
&\qquad\qquad \in [\![f(1)]\!] \circ [\![x]\!]\}.
\end{aligned} \tag{5}$$

Now by 3.7(vi) and 3.6(i), we have

$$\begin{aligned}
[\![f(1)]\!] &= \{(\sigma, \mathsf{rest}(\tau)) \mid \exists(\lambda x.p, \beta) \\
&\qquad\qquad \sigma(f) = (\lambda x.p, \beta),\ ((x = 1 : \beta) + \sigma, \tau) \in [\![p]\!]\} \\
&\quad \cup\ \{(\sigma, \sigma) \mid \sigma(f) \text{ exists and is intrinsic}\} \\
[\![x]\!] &= \{(\sigma, \sigma, \sigma(x)) \mid \sigma(x) \text{ exists}\}.
\end{aligned}$$

Composing these two relations and using the distributivity of composition over union, we have

$$\begin{aligned}
[\![f(1)]\!] \circ [\![x]\!] = \{(\sigma, \mathsf{rest}(\tau), \mathsf{rest}(\tau)(x)) \mid\ & \exists(\lambda x.p, \beta)\ \sigma(f) = (\lambda x.p, \beta), \\
& ((x = 1 : \beta) + \sigma, \tau) \in [\![p]\!], \\
& \mathsf{rest}(\tau)(x) \text{ exists}\} \\
\cup\ \{(\sigma, \sigma, \sigma(x)) \mid\ & \sigma(f) \text{ exists and is intrinsic},\ \sigma(x) \text{ exists}\}.
\end{aligned}$$

Combining this with (5) and simplifying yields

$$\begin{aligned}
&[\![\lambda f.(f(1)\,;\; x)\;\;\lambda z.(y := y + z\,;\; x := y)]\!] \\
&= \{(\theta, \mathsf{rest}(\eta), b) \mid \exists\rho\ \exists\tau\ \eta = \mathsf{rest}(\tau),\ b = \mathsf{rest}(\tau)(x), \\
&\qquad\qquad \rho = (f = (\lambda z.(y := y + z\,;\; x := y), \mathsf{actv}(\theta))) + \theta, \\
&\qquad\qquad ((z = 1 : \mathsf{actv}(\theta)) + \rho, \tau) \in [\![y := y + z\,;\; x := y]\!]\}.
\end{aligned} \tag{6}$$

Using 3.7(i) and (iv),

$$\begin{aligned}
[\![y := y + z]\!] &= \{(\sigma, \sigma[y/\sigma(y) + \sigma(z)]) \mid \sigma(y),\ \sigma(z) \text{ exist}\} \\
[\![x := y]\!] &= \{(\sigma, \sigma[x/\sigma(y)]) \mid \sigma(x),\ \sigma(y) \text{ exist}\} \\
[\![y := y + z\,;\ x := y]\!] &= [\![y := y + z]\!] \circ [\![x := y]\!] \\
&= \{(\sigma, \sigma[y/\sigma(y) + \sigma(z)][x/\sigma(y) + \sigma(z)]) \mid \sigma(x),\ \sigma(y), \\
&\qquad\qquad \sigma(z) \text{ exist}\}.
\end{aligned}$$

Using this, the last condition of (6) simplifies to

$$\begin{aligned}
&((z = 1 : \mathsf{actv}(\theta)) + \sigma, \tau) \in [\![y := y + z\,;\ x := y]\!] \\
&\Leftrightarrow \tau = ((z = 1 : \mathsf{actv}(\theta)) + \sigma)[y/\theta(y) + 1][x/\theta(y) + 1]),\ \theta(x),\ \theta(y) \text{ exist.}
\end{aligned}$$

Plugging this into (6) and simplifying further, we obtain

$$\begin{aligned}
&[\lambda f.(f(1)\,;\ x)\ \ \lambda z.(y := y + z\,;\ x := y)] \\
&= \{(\theta, \theta[y/\theta(y) + 1][x/\theta(y) + 1], \theta(y) + 1) \mid \theta(x),\ \theta(y) \text{ exist}\}.
\end{aligned}$$

This allows us to simplify the last condition of (4):

$$\begin{aligned}
&((y = 4) + \sigma, \tau, b) \in [\lambda f.(f(1)\,;\ x)\ \ \lambda z.(y := y + z\,;\ x := y)] \\
&\Leftrightarrow \tau = (y = 5) : (\sigma[x/5]),\ b = 5,\ \sigma(x) \text{ exists.}
\end{aligned}$$

Finally, plugging this back into (4) and simplifying, we obtain the desired result:

$$\begin{aligned}
&[\lambda y.(\lambda f.(f(1)\,;\ x)\ \ \lambda z.(y := y + z\,;\ x := y))\ \ (4)] \\
&= \{(\sigma, \sigma[x/5], 5) \mid \sigma(x) \text{ exists}\}.
\end{aligned}$$

Although this calculation is much abbreviated, we have used nothing beyond elementary logic, set theory, and relational algebra, along with a few self-evident properties of closure structures.

4 Relational Semantics in Program Equivalence Proofs

In this section we prove six of the seven equivalences of Meyer and Sieber [12].

We begin with a general bisimulation result. Let $\sigma, \widehat{\sigma}$ be closure structures. Let $f : \sigma \to \widehat{\sigma}$ be a function mapping nodes in σ to nodes in $\widehat{\sigma}$ and stack entries in σ to stack entries in $\widehat{\sigma}$. We say that f *embeds* σ *in* $\widehat{\sigma}$ if

- f is one-to-one on both nodes and stack entries,
- $f(\mathsf{actv}(\sigma)) = \mathsf{actv}(\widehat{\sigma})$,
- f preserves stack order,
- f preserves all reference relationships and node labels in the following sense:
  - $f(\mathsf{parent}(\alpha)) = \mathsf{parent}(f(\alpha))$,
  - $f(\mathsf{root}(\sigma)) = \mathsf{root}(\widehat{\sigma})$,

- if i is a stack entry of σ containing a reference to α, then $f(i)$ contains a reference to $f(\alpha)$,
- if the node α contains the binding $x = c$ and c is an intrinsic value, then $f(\alpha)$ contains $x = c$,
- if the node α contains a binding to a closure $x = (t, \beta)$, then $f(\alpha)$ contains $x = (t, f(\beta))$.

Thus $\widehat{\sigma}$ contains an isomorphic copy of σ, possibly with some extra stack entries and accessible nodes. However, the subtree of σ consisting of nodes accessible from the active environment is isomorphic to that of $\widehat{\sigma}$, and this determines all computational behavior from those input states. This intuition is captured in the following bisimulation property.

Lemma 1. *Suppose f embeds σ in $\widehat{\sigma}$. Let p be a program expression.*

(i) *If $(\sigma, \tau) \in [\![p]\!]$, then there exist $\widehat{\tau}$ and f' such that $(\widehat{\sigma}, \widehat{\tau}) \in [\![p]\!]$ and f' embeds τ in $\widehat{\tau}$.*
(ii) *If $(\widehat{\sigma}, \widehat{\tau}) \in [\![p]\!]$, then there exist τ and f' such that $(\sigma, \tau) \in [\![p]\!]$ and f' embeds τ in $\widehat{\tau}$.*

Moreover, in both cases f and f' agree on the stack (recall from Section 3.8 that the stacks of σ and τ are the same, as are the stacks of $\widehat{\sigma}$ and $\widehat{\tau}$).

Proof. The proof is by induction on p, with the induction hypothesis suitably strengthened to include $[\![e]\!]$ for value expressions. We argue (i) for cases 3.7(i) and (vii) explicitly.

For 3.7(i), suppose $(\sigma, \tau) \in [\![x := a]\!]$. Then there exist ρ and c such that $(\sigma, \rho, c) \in [\![a]\!]$, $\sigma(x)$ exists, and $\tau = \rho[x/c]$. Then $\widehat{\sigma}(x)$ exists, since σ and $\widehat{\sigma}$ have isomorphic active environments. By the induction hypothesis on a, there exist $\widehat{\rho}$ and an embedding $f' : \rho \to \widehat{\rho}$ such that $(\widehat{\sigma}, \widehat{\rho}, \widehat{c}) \in [\![a]\!]$, where $\widehat{c} = c$ if c is an intrinsic value, and if c is a closure (t, β), then $\widehat{c} = (t, f'(\beta))$. Letting $\widehat{\tau} = \widehat{\rho}[x/\widehat{c}]$, we have that f' embeds τ in $\widehat{\tau}$ and $(\widehat{\sigma}, \widehat{\tau}) \in [\![x := a]\!]$.

For 3.7(vii), suppose $(\sigma, \tau) \in [\![P()]\!]$. Then there exist ρ and υ such that $(\sigma, \rho, (\lambda().p, \beta)) \in [\![P]\!]$ (say), $(\beta + \rho, \upsilon) \in [\![p]\!]$, and $\tau = \mathsf{rest}(\upsilon)$. By the induction hypothesis on P, there exist $\widehat{\rho}$ and embedding $f' : \rho \to \widehat{\rho}$ such that $(\widehat{\sigma}, \widehat{\rho}, (\lambda().p, f'(\beta))) \in [\![P]\!]$. Form the new closure structure $f'(\beta) + \widehat{\rho}$ and extend f' to an embedding $\beta + \rho \to f'(\beta) + \widehat{\rho}$ (the extension is uniquely determined), which we still denote it by f'. By the induction hypothesis on p, there exist $\widehat{\upsilon}$ and embedding $f'' : \upsilon \to \widehat{\upsilon}$ such that $(f'(\beta) + \widehat{\rho}, \widehat{\upsilon}) \in [\![p]\!]$. Defining $\widehat{\tau} = \mathsf{rest}(\widehat{\upsilon})$, we have $(\widehat{\sigma}, \widehat{\tau}) \in [\![P()]\!]$ and f'' an embedding of τ in $\widehat{\tau}$. □

The first two examples of Meyer and Sieber examine the inability of procedures to access variables not in scope at the time of their declaration.

Example 1. For a procedure identifier P of type $\mathsf{void} \to \mathsf{void}$, x distinct from P, and c a constant, the following two programs are equivalent.

$$\mathsf{let}\ x = c\ \mathsf{in}\ P()\ \mathsf{end} \qquad P().$$

Proof. From 3.6(iii) and (iv), after simplification we have

$$\begin{aligned}[\![\mathsf{let}\ x = c\ \mathsf{in}\ P()\ \mathsf{end}]\!] &= [\![\lambda x.P()\ \ c]\!] \\ &= \{(\sigma, \mathsf{rest}(\tau)) \mid ((x = c) + \sigma, \tau) \in [\![P()]\!]\}. \qquad (7)\end{aligned}$$

Similarly, from 3.7(vii) and 3.6(i), we have

$$\begin{aligned}[\![P()]\!] = \{(\sigma, \mathsf{rest}(\tau)) \mid \sigma(P) = (\lambda().p, \beta),\ (\beta + \sigma, \tau) \in [\![p]\!]\} \qquad (8)\\ \cup\ \{(\sigma, \sigma) \mid \sigma(P) \text{ exists and is an intrinsic value}\}.\end{aligned}$$

Substituting (8) in (7) and simplifying, we obtain

$$\begin{aligned}&[\![\mathsf{let}\ x = c\ \mathsf{in}\ P()\ \mathsf{end}]\!] \\ &= \{(\sigma, \mathsf{rest}(\mathsf{rest}(\eta)))\ \mid\ \sigma(P) = (\lambda().p, \beta),\ (\beta + (x = c) + \sigma, \eta) \in [\![p]\!]\} \qquad (9)\\ &\quad \cup\ \{(\sigma, \sigma) \mid \sigma(P) \text{ exists and is an intrinsic value}\}.\end{aligned}$$

To show (8) and (9) are equal, it suffices to show that for all ρ, the following two statements are equivalent:

$$\begin{aligned}\exists\eta\ \rho = \mathsf{rest}(\mathsf{rest}(\eta)),\ (\beta + (x = c) + \sigma, \eta) \in [\![p]\!],\\ \exists\tau\ \rho = \mathsf{rest}(\tau),\ (\beta + \sigma, \tau) \in [\![p]\!].\end{aligned}$$

This follows directly from Lemma 1 once we have constructing an embedding $\beta + \sigma \to \beta + (x = c) + \sigma$. The embedding is the identity on the tree of σ and maps the stack elements of $\beta + \sigma$ to the stack elements of $\beta + (x = c) + \sigma$ in order, but skipping the top element, which is $\mathsf{actv}((x = c) + \sigma)$. □

Example 2. For a procedure identifier P of type $\mathsf{void} \to \mathsf{void}$, x distinct from P, and c a constant, the following two programs are equivalent.

$\mathsf{let}\ \ x = c$	$\mathsf{let}\ \ x = c$
$\mathsf{in}\ \ P()\ ;\ u$	$\mathsf{in}\ \ P()\ ;\ (x = c)\ ;\ u$
end	end

Proof. The equation asserts that the test $x = c$ is redundant after the evaluation of $P()$. The proof is similar to that of Example 1. After expanding the definitions and simplifying, it comes down to showing that if $\sigma(P) = (\lambda().p, \beta)$, and if $(\beta + (x = c) + \sigma,\ \rho) \in [\![p]\!]$, then $\rho(x) = c$. This follows from Lemma 1 by constructing an embedding of $\beta + \sigma$ in $\beta + (x = c) + \sigma$, giving a bisimilar computation that cannot change the value of x. □

The next example demonstrates that the effect of a function does not depend on the names of the arguments. This is a feature of the call-by-value parameter passing mechanism.

Example 3. Let x, y, and Q be distinct variables and b, c constants. The following two programs are equivalent:

$\mathsf{let}\ \ x = b, y = c$	$\mathsf{let}\ \ x = c, y = b$
$\mathsf{in}\ \ Q(x)(y)$	$\mathsf{in}\ \ Q(y)(x)$
end	end

Proof. It suffices to show that both programs are equivalent to $Q(b)(c)$; that is, this is an instance of when call-by-value and call-by-name (β-reduction) give the same result. We can do this in stages: to show the first program is equivalent to $Q(b)(c)$, it suffices to show that

$$\begin{aligned}\mathsf{let}\ y = c\ \mathsf{in}\ Q(x)(y)\ \mathsf{end} &= Q(x)(c)\\ \mathsf{let}\ x = b\ \mathsf{in}\ Q(x)(c)\ \mathsf{end} &= Q(b)(c).\end{aligned}$$

Let us argue the former.

Suppose both Q and x are defined in σ, say $\sigma(Q) = (\lambda z.q; e,\ \beta)$ and $\sigma(x) = b$. To calculate $[\![\mathsf{let}\ y = c\ \mathsf{in}\ Q(x)(y)\ \mathsf{end}]\!]$, we expand the definition and simplify. Prepending the binding $y = c$ to $\mathsf{actv}(\sigma)$ and evaluating $Q(x)$ in that environment, we would get (say)

$$((z = b : \beta) + (y = c) + \sigma,\ \rho,\ (\lambda w.p,\ \gamma)) \in [\![q; e]\!], \tag{10}$$

and we wish to apply $(\lambda w.p,\ \gamma)$ to y in the active environment of $\mathsf{rest}(\rho)$.

The corresponding calculation for $[\![Q(x)(c)]\!]$ starts in state $(z = b : \beta) + \sigma$. But there is an embedding of this state in $(z = b : \beta) + (y = c) + \sigma$ that omits the top stack element containing the binding $y = c$. By Lemma 1, we have

$$((z = b : \beta) + \sigma,\ \upsilon,\ (\lambda w.p,\ \gamma)) \in [\![q; e]\!] \tag{11}$$

with an embedding $f : \upsilon \to \rho$, and we wish to apply $(\lambda w.p,\ \gamma)$ to c in the active environment of $\mathsf{rest}(\upsilon)$. Now f restricted to $\mathsf{rest}(\upsilon)$ is not an embedding in $\mathsf{rest}(\rho)$, since the active environment is not mapped correctly; but it is an embedding in $\mathsf{rest}(\mathsf{rest}(\rho))$. Moreover, the stack sizes of $\mathsf{rest}(\upsilon)$ and $\mathsf{rest}(\mathsf{rest}(\rho))$ are the same, so the embedding is an isomorphism. This says that $\mathsf{rest}(\mathsf{rest}(\rho)) = \mathsf{rest}(\upsilon)$. Furthermore, the value of y was not changed in (10), since there is a bisimilar computation (11) in which it was not changed. This says that

$$\mathsf{rest}(\rho) = (y = c) + \mathsf{rest}(\upsilon).$$

It remains to argue that the final application of $(\lambda w.p,\ \gamma)$ yields the same result in both cases. Again we have an embedding and can apply Lemma 1. The two expressions are

$$\begin{aligned}((w = c : \gamma) + (y = c) + \mathsf{rest}(\upsilon),\ \theta) &\in [\![p]\!]\\ ((w = c : \gamma) + \mathsf{rest}(\upsilon),\ \eta) &\in [\![p]\!]\end{aligned}$$

with an embedding $f : \eta \to \theta$. The final output states are $\mathsf{rest}(\mathsf{rest}(\theta))$ and $\mathsf{rest}(\eta)$, which are isomorphic because f embeds $\mathsf{rest}(\eta)$ in $\mathsf{rest}(\mathsf{rest}(\theta))$ and the stack sizes are the same. □

The remaining examples look at the higher-order case in the presence of local variables. The goal of these examples is to prove that procedures that have as arguments procedures with private data cannot access that private data.

In this example, we look at a procedure with two local variables that only alters one of them.

Example 4. For distinct variables x, y, Q, and T, the following two programs are equivalent:

let $x = 0,\ y = 1,\ T = \lambda().y := 2y$	let $x = 0,\ y = 1,\ T = \lambda().y := 2y$
in $Q(T)\ ;\ (x = 0)\ ;\ u$	in $Q(T)\ ;\ u$
end	end

Proof. Let

$$\begin{aligned}\rho &= (y = 1) + (x = 0) + \sigma,\\ \gamma &= \mathsf{actv}(\rho),\\ \xi &= (T = (\lambda().y := 2y, \gamma) + (y = 1) + (x = 0) + \sigma.\end{aligned}$$

Starting in state σ, ξ is the state after binding x, y, and T in the let expression. Suppose $\sigma(Q) = (\lambda R.p,\ \beta)$. This is also $\xi(Q)$. After substituting the definitions and simplifying, the proof comes down to showing that if

$$((R = (\lambda().y := 2y, \gamma) : \beta) + \xi,\ \eta) \in [\![p]\!],$$

then $\eta(x) = 0$. By Lemma 1, removing all stack elements, there is a bisimilar computation

$$((R = (\lambda().y := 2y, \gamma) : \beta) + \varepsilon,\ \theta) \in [\![p]\!],$$

where ε is an abbreviation for the empty closure structure. As β is a node of σ, the only reference to the binding $x = 0$ in this closure structure is via the closure $(\lambda().y := 2y, \gamma)$. All that can be done with this object is to apply it or assign it to a variable, and neither operation changes the value of x or changes the fact that the only reference to $x = 0$ is via γ in the closure. This is clear for assignments. An application $R()$ in a state τ in which R is bound to $(\lambda().y := 2y, \gamma)$ yields output state $\mathsf{rest}(\upsilon)$, where $(\gamma + \tau, \upsilon) \in [\![y := 2y]\!]$. The value of x is unchanged due to the form of the assignment, and the reference to γ on the stack during this calculation is transitory. □

In the next example, we want to know that if an invariant on a local variable is maintained by a function, then that invariant is maintained for the entire program if the variable is only accessed through that function.

Example 5. For distinct variables x, Q, and A_2, the following two programs are equivalent:

let $x = 0, A_2 = \lambda().x := x + 2$	let $x = 0, A_2 = \lambda().x := x + 2$
in $Q(A_2)\ ;\ (x \bmod 2 = 0)\ ;\ u$	in $Q(A_2)\ ;\ u$
end	end

Proof. This example is very similar to the previous. In this case, we note that if $\sigma(x) \bmod 2 = 0$ and $(\sigma, \tau) \in [\![A_2]\!]$, then $\tau(x) \bmod 2 = 0$. □

The final example demonstrates that the behavior of a procedure is not affected by the values of another procedure's local variables.

Example 6. Let x, Q, A_1, and A_2 be distinct variables. The following two programs are equivalent:

let $x = 0, A_1 = \lambda().x := x + 1$	let $x = 0, A_2 = \lambda().x := x + 2$
in $Q(A_1)$	in $Q(A_2)$
end	end

Proof. In this example, it is important to note that A_1 and A_2 have void return type. The argument is very similar to the argument in Example 5. We use Lemma 1 to obtain bisimilar computations in which x is not accessible except via the closures bound to A_1 and A_2, therefore can only be altered by calls to A_1 and A_2. The execution of Q is always in a preexisting environment with no other access to x. Finally, the bindings of x, A_1 and A_2 are discarded at the end, leaving equivalent output states. □

The one example from Meyer and Sieber that our system cannot currently handle deals with the inability of local variables to be aliased by variables declared elsewhere. We currently have neither the means to alias a location nor to test for aliasing.

5 Conclusion and Future Work

We have presented a compositional relational semantics that captures all contextual information in the state, allowing us to reason about programs with local state in an equational way without consideration of context. We have shown how to reason in this framework by proving several benchmark examples of Meyer and Sieber [12].

While we do not deal with the more intricate issue of aliasing, there is no reason to believe our approach could not be extended to do so. We are currently attempting to expand the definition of closure structure to allow explicit references as values.

Using relational semantics for higher-order programs does not solve problems that many other methods cannot, it simply allows one to reason in a natural equational style that is mathematically based, yet true to the underlying operational intuition.

6 Acknowledgments

We are grateful to Jules Desharnais, Matthew Fluet, Riccardo Pucella, and three anonymous reviewers 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. Milne, R., Strachey, C.: A Theory of Programming Language Semantics. Halsted Press, New York, NY, USA (1977)
5. Scott, D.: Mathematical concepts in programmng language semantics. In: Proc. 1972 Spring Joint Computer Conferneces, Montvale, NJ, AFIPS Press (1972) 225–34
6. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA (1981)
7. 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
8. Stark, I.: Categorical models for local names. LISP and Symbolic Computation **9**(1) (1996) 77–107
9. Reyonlds, J.: The essence of ALGOL. In de Bakker, J., van Vliet, J.C., eds.: Algorithmic Languages, North-Holland, Amsterdam (1981) 345–372
10. Oles, F.J.: A category-theoretic approach to the semantics of programming languages. PhD thesis, Syracuse University (1982)
11. 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
12. 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
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. Riecke, J.G., Sandholm, A.: A relational account of call-by-value sequentiality. In: Proc. 12th Symp. Logic in Computer Science (LICS'97). (1997) 258–267
24. Riecke, J.G., Viswanathan, R.: Isolating side effects in sequential languages. In: Proc. 22th Symp. Principles of Programming Languages (POPL'95). (1995) 1–12
25. Fiore, M.P., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. 14th Symp. Logic in Computer Science (LICS'99). (1999) 193–202
26. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge, MA (2000)
27. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems **19**(3) (1997) 427–443
28. Abadi, M., Cardelli, L.: A Theory of Objects. Springer (1996)

Appendix Skeleton Implementation of Closure Structures

For illustrative purposes, we provide here a skeleton of a simple untyped implementation of closure structures in ML that is faithful to the description in Section 3. The type `cs` represents the active environment and stack of a closure structure; the tree is implicit. The second component of a `binding` is declared as a reference to allow destructive updates.

```
type var = string
type lambdaExpr = string

datatype value = Int of int | Closure of lambdaExpr * environment
withtype binding = var * value ref
and environment = binding list

type cs = environment * environment list

fun newCS () : cs = ([],[])

fun lookup (v:var) ((act,s):cs) : value option =
  let fun lookup' (v:var) (env:environment) : value option =
    case env of [] => NONE
        | (u,c)::t => if u=v then SOME (!c) else lookup' v t
  in lookup' v act
  end

fun update (v:var) (d:value) ((act,s):cs) : unit =
  let fun update' (v:var) (d:value) (env:environment) : unit =
    case env of [] => ()
        | (u,c)::t => if u=v then c := d else update' v d t
  in update' v d act
  end

fun createBinding (v:var) (c:value) ((act,s):cs) : cs =
  let val b = (v,ref c) : binding
      val env = b::(hd s) : environment
  in (env, act::(tl s))
  end
```