# Denotational Semantics Using an Operationally-Based Term Model


Mitchell Wand* Gregory T. Sullivan*
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
voice: (617) 373-2462, fax: (617) 373-5121
{wand,gregs}@ccs.neu.edu
http://www.ccs.neu.edu/home/{wand,gregs}



## Abstract

We introduce a method for proving the correctness of transformations of programs in languages like Scheme and ML. The method consists of giving the programs a denotational semantics in an operationally-based term model in which interaction is the basic observable, and showing that the transformation is meaning-preserving. This allows us to consider correctness for programs that interact with their environment without terminating, and also for transformations that change the internal store behavior of the program. We illustrate the technique on one of the Meyer-Sieber examples, and we use it to prove the correctness of assignment elimination for Scheme. The latter is an important but subtle step for Scheme compilers; we believe ours is the first proof of its correctness.


## 1 Introduction

Compilers for higher-order languages typically perform elaborate program transformations in order to improve performance. Such transformations often change the storage behavior of the program. For concreteness, let us consider a simple language that, like Scheme, allocates a new mutable cell for each parameter on each procedure call. Then we should have an equivalence

$$((\mathsf{lambda}\ (x)\ x)\ e) \equiv e \tag{1}$$

even though the left-hand program allocates one more cell than the right-hand program. We should expect a good compiler to perform this optimization, and we would expect the semantics of the language to justify it.


*Work supported by the National Science Foundation under grants number CCR-9304144 and CCR-9404646. Current address: Gregory T. Sullivan, Fulton 414, Computer Science Department, Boston College, Chestnut Hill, MA 02167. E-mail: gregs@cs.bc.edu ; WWW: http://www.cs.bc.edu/~gregs .

To appear in POPL '97.


Unfortunately, in the denotational semantics of Scheme, the equivalence above is unsound, because these programs could be executed in a continuation that tests the store in ways that are not expressible in Scheme itself. It is possible to construct denotational semantics in which some equivalences like (1) are true, but they are quite complex, and none extend far enough to encompass real languages [MS88, Ole85, OT95, Ode94].

We seek to prove the correctness of such source-level transformations and translations in compilers for languages like Scheme and ML. This setting is challenging in a number of respects:

- We seek methods that are applicable to both untyped and typed languages, including Scheme, ML, and Idealized Algol, and also to translations from one language to another.
- We seek methods that will be sound even for reasoning about programs like servers that do not terminate.
- We seek methods that will be sound even for transformations that change the behavior of a heap-allocated store.

Our approach to such equivalences is to give the language a denotational semantics in an *event-based model.* An event-based model is a term model embedded in a process calculus. Observable behavior is taken from the process calculus; equality of terms in the model is taken to be contextual equivalence as measured by this observable behavior.

We illustrate the technique on one of the Meyer-Sieber examples [MS88], and we use it to prove the correctness of assignment elimination, an important optimization used in compilers for Scheme [KKsR$^+$86].

We believe that this paper makes two contributions: first, it formulates a framework that handles all the requirements above. Second, it presents the first known proof of correctness of assignment elimination.

Following a review of related work in Section 2, we begin, in Section 3, by defining a metalanguage. This metalanguage is an extended PCF with recursive types, a store, and a CPS I/O system. The semantics of our source languages will be specified by syntax-directed translations into this metalanguage. The recursive types in the metalanguage allow us to translate both typed and untyped languages easily.

In Sections 3.2 and 3.3, we define the behavior of a metalanguage term, and define two terms to be *contextually equivalent* (denoted $\cong_{\forall C}$) if they have the same I/O behavior when inserted into every possible program context. By identifying input/output as the observable actions of the system, we avoid considering the store as part of the "final answer" of a computation. Indeed, we avoid the notion of a "final answer" at all, so that we can study equivalence of terms that have infinite behavior [CG95, TW96].

Contextual equivalence is, by the form of its definition, a *congruence* – that is, it is preserved by all term constructors of the language. Furthermore, it is easy to show (using the confluence property for PCF), that if $M =_{pcf} N$, then $M \cong_{\forall C} N$. Hence the structure obtained by taking equivalence classes of open terms under $\cong_{\forall C}$ is a model of typed lambda-calculus. This is our operationally-based term model. Because this is a model, ordinary reasoning using $\beta$-conversion is valid in the model. This distinguishes it from treatments like [MT91], in which only the weaker $\beta_v$ rule is valid.

Unfortunately, the quantification over all contexts makes $\cong_{\forall C}$ difficult to establish directly. We must consider not only contexts in which the terms are executed, but also contexts that pass the terms around, put many copies of them in the store, etc. Our major tool for easing this proof burden is a "context lemma" (Theorem 4.2) stating that two terms are contextually equivalent if and only if they are equivalent in a much smaller set of contexts we call *extensional contexts*; for types constructed with $\rightarrow$, these are the same as head contexts.

We then turn to the specification of source languages. We can specify a semantics in this model by giving a compositional (syntax-directed) translation $[\![-]\!]_{\mathcal{L}}$ from a source language $L$ to terms of the metalanguage; the meaning of a source language expression is the $\cong_{\forall C}$-equivalence class of the translated term. To show two source language expressions $e_1$ and $e_2$ have the same meaning, we must show that their translations are equal, i.e., that $[\![e_1]\!]_{\mathcal{L}} \cong_{\forall C} [\![e_2]\!]_{\mathcal{L}}$ To show a translation $\Phi$ from $L$ to $L'$ is correct, we equip $L'$ with a similar translation $[\![-]\!]_{\mathcal{L}'}$ and show that $[\![e]\!]_{\mathcal{L}} \cong_{\forall C} [\![\Phi(e)]\!]_{\mathcal{L}'}$.

Our translations will produce terms in continuation-passing style. This means that the translation will not be fully abstract for stack-based languages like Idealized Algol. However, we are primarily concerned with languages like Scheme and ML that already have continuations, and CPS translation is well-suited to these languages.

Section 5 illustrates this technique using an example from [MS88].

In Section 6, we use the technique to prove the correctness of the assignment elimination transformation for Scheme [KKsR+86]. We conclude with consideration of open problems and future work.

## 2 Related Work

Operationally-based term models were introduced in [TW96]. Here we extend [TW96] by adding types and a heap, and by considering applications to program transformations. The formulation of input-output both here and in [TW96] is based on Gordon's CPS input-output system [Gor94].

Observational or contextual equivalence for programming languages was introduced by [Plo77], adapting the work of Morris [Mor68] for the $\lambda$-calculus. Plotkin's formulation, like almost all other work for sequential languages, dealt with conversion to a constant. Context lemmas for proving contextual equivalence were introduced by Milner [Mil77] in a typed functional language. Gordon [Gor95] extended the result to include recursive types with potentially infinite behavior.

The work most closely related to ours is that of Mason and Talcott [MT91] and Felleisen [Fel87]. They considered contextual equivalence for a number of untyped call-by-value languages using operational techniques, and proved context lemmas for these languages. Our approach differs from theirs in that we propose a single metalanguage into which a variety of source languages can be translated; neither Mason and Talcott nor Felleisen consider issues of translation. Our Context Lemma (Theorem 4.2) also extends their results by considering input-output effects rather than final answers and by validating full $\beta$-conversion rather than their $\beta$-value.

Crole and Gordon [CG95] give a denotational semantics for a typed call-by-value language with input-output by translation into a metalogic $\mathcal{M}$ similar in purpose, though not in structure, to our metalanguage. Their translation uses a monadic style in place of our continuation-passing style. Their treatment does not include a store, and it is not clear whether their system can be extended to deal with latently recursively typed languages like Scheme.

Most work on store semantics [MS88, Ole85, OT95, Ode94] deals with stores in stack-discipline languages, and are complicated by the need to model that restriction. Most of this work depends crucially on the type structure of the language; our work is applicable to untyped or latently typed languages as well. [PS93] treats heap-allocated names without content or mutation.

Bisimulation was introduced by [Par81]. Abramsky [Abr90] used bisimulation to define a $\lambda$-model, but his observables were very different from ours. The concept of a relation preserved under computation is probably much older. It was used for program equivalence in [MT91, WO92, ORW95]; [Sta94] uses a similar notion extended to become a logical relation.

Assignment elimination was introduced in the Orbit compiler [KKsR+86] and used in [KH89, CH94]. It was omitted from [GSR95] because the authors were unable to prove its correctness [Ram96].

## 3 An Operationally-Based Term Model

### 3.1 Syntax

Values in the model are equivalence classes of terms in an extended $\lambda$-calculus that we call the *metalanguage*.

The types in the metalanguage, ranged over by $\tau$, are potentially infinite trees which respect the following structure:

$$o ::= unit \mid bool \mid int \mid ref(\tau) \mid pr$$
$$\tau ::= o \mid \tau{\rightarrow}\tau \mid \tau{+}\tau \mid \tau{\times}\tau$$

The base types $o$ include *int* for integers, *bool* for booleans, $ref(\tau)$ for locations with contents of type $\tau$, and *unit* for the singleton type. Type *pr* is for processes, which may do I/O.

We allow infinite types, so we will need to define operations on types coinductively instead of inductively. We write $\tau^*$ for the infinite type

$$unit + (\tau \times (unit + (\tau \times \ldots)))$$

of lists of elements of type $\tau$.

The syntax of our metalanguage is given by the following grammar:

$$M ::= x \mid n \mid c \mid l^\tau \mid \lambda x{:}\tau.M \mid MM$$

with $x$ ranging over a countably infinite set of variables and $l$ ranging over a countably infinite set of location constants. Locations $l^\tau$ are annotated with types and are assigned type $ref(\tau)$. The typing rules for the language are standard. The meta-operation $locs(M)$ returns the set of locations occurring in $M$ and is easily defined via structural induction.

The metalanguage comes equipped with PCF-like constants and the usual constants for sums and products. More interesting are the constants for stores and processes, whose types are given in Figure 1. Their behavior is considered in Section 3.2.

In the remainder of this section, we turn this language into a model of the lambda-calculus by defining the basic observable to be the input-output behavior of a closed term of type *pr*. We then define two terms to be equivalent ($\cong_{\forall C}$) iff they have the same observable behavior in any closing context $C[-]$ of type *pr*. It follows immediately that $\cong_{\forall C}$ is a congruence that respects $\beta$-reduction, so the equivalence classes of metalanguage terms under $\cong_{\forall C}$ form a model of typed lambda-calculus.

### 3.2 Operational Semantics and Observable Equivalence

We define the operational semantics of the metalanguage in three steps. The reduction semantics of the PCF fragment of the metalanguage defines a relation $\rightarrow_{pcf}$ between closed well-typed terms. The reduction relation $\rightarrow_{sto}$ is defined on states, includes PCF reduction within term components, and gives transition rules for states with store operations in head position. Finally, we define a labeled transition relation, $\xrightarrow{a}_{io}$, between states which reduce under $\rightarrow^*_{sto}$ to states headed by I/O operations.

The definition of $\rightarrow_{pcf}$ is the usual call-by-name reduction [Plo77] and is omitted. Figure 2 defines the transition relation $\rightarrow_{sto}$ between states. A *state*, denoted $\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle$, is a term-store pair where $M$ is a closed term of type *pr* and store $\Sigma$ is a finite map from locations to terms. All locations $l^\tau$ in the domain of $\Sigma$ must map to closed terms of type $\tau$.

The first rule of Figure 2 incorporates PCF reduction into $\rightarrow_{sto}$; if the term component of a state may take a single PCF transition, the state pair may take the corresponding transition. The new operator allocates a new location in the store, initializes the fresh location to its first argument, and invokes its second argument with the new location. The deref operator invokes its second argument with the value in the store at the location given by its first argument. The update operator destructively updates the location given by its first argument to be the term given by its second argument and then invokes its third argument. The reflexive, transitive closure of $\rightarrow_{sto}$ is denoted by $\rightarrow^*_{sto}$.

A labeled transition system $\xrightarrow{a}_{io}$ for the I/O operators is defined in Figure 3. The write operator writes its integer argument and invokes its continuation. The read operator reads an integer and invokes its continuation with the number read. We make termination observable by having a stop operator that takes a $\surd$ transition to a stopped state $\langle\!\langle \mathsf{stopped}\ ;\ \Sigma'\rangle\!\rangle$, from which there are no transitions.

Our notion of observable equivalence on states is $\approx_{sim}$, the standard notion of (strong) bisimulation for this labelled transition system.[1]

It is easy to show that consistent renaming of locations preserves observable equivalence. Two states are equal up to a renaming of location constants, denoted $\alpha_{loc}$, if there is a relation $\Theta$ between locations satisfying the following properties:

1. if $(i,j) \in \Theta$ and $(i',j') \in \Theta$, then $(i = i' \Leftrightarrow j = j')$
2. $M\Theta = N$
3. $locs(M) \subseteq dom(\Theta)$
4. if $(l,l') \in \Theta$ then either
   (a) $l \notin dom(\Sigma_M)$ and $l \notin dom(\Sigma_N)$ or
   (b) $\Sigma_M(l)\Theta = \Sigma_N(l')$ and $locs(\Sigma_M(l)) \subseteq dom(\Theta)$

**Lemma 3.1**

$$\alpha_{loc} \subseteq \lesssim_{sim}$$

PROOF SKETCH: Use coinduction on the definition of $\approx_{sim}$.

[1] Alternatively, we could have made $\rightarrow_{sto}$ into a silent transition, and used weak bisimulation. We believe the results would be the same; the proofs would look somewhat different.

| | |
|---|---|
| $\mathsf{write}: int \to pr \to pr$, | write an integer to stdout |
| $\mathsf{read}: (int \to pr) \to pr$, | read integer from stdin |
| $\mathsf{stop}: pr$, $\mathsf{stopped}: pr$, | halt, halted |
| $\mathsf{new}^\tau : \tau \to (ref(\tau) \to pr) \to pr$, | get fresh loc, initialize to 1st arg |
| $\mathsf{deref}^\tau : ref(\tau) \to (\tau \to pr) \to pr$, | dereference a location |
| $\mathsf{update}^\tau : ref(\tau) \to \tau \to pr \to pr$, | update a location |
| $\mathsf{eq?}_{loc}{}^\tau : ref(\tau) \to ref(\tau) \to bool$ | compare locations for equality |

Figure 1: Metalanguage Constants for Store and I/O Operations

$$
\begin{array}{ll}
M \rightarrow_{pcf} M' \Rightarrow \langle\!\langle M \,;\, \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle M' \,;\, \Sigma \rangle\!\rangle & \\
\langle\!\langle \mathsf{new}^\tau\, M N \,;\, \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle N l^\tau \,;\, \Sigma[l^\tau \mapsto M] \rangle\!\rangle & \text{if } l^\tau \notin dom(\Sigma) \cup locs(rng(\Sigma), M, N) \\
\langle\!\langle \mathsf{deref}^\tau\, l^\tau M \,;\, \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle M N \,;\, \Sigma \rangle\!\rangle & \text{if } \Sigma(l^\tau) = N \\
\langle\!\langle \mathsf{update}^\tau\, l^\tau M N \,;\, \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle N \,;\, \Sigma[l^\tau \mapsto M] \rangle\!\rangle & \text{if } l^\tau \in dom(\Sigma)
\end{array}
$$

Figure 2: Operational Semantics for Store Operators

### 3.3 Term Equivalence

We say that two terms are *contextually equivalent* if they have the same I/O behavior when inserted into every possible program context:[2]

**Definition 3.2 (contextual equivalence, $\cong_{\forall C}$)** *If $\Gamma \vdash M, N{:}\tau$, then $\Gamma \vdash M \cong_{\forall C} N{:}\tau$ iff for all closing contexts $C[-_\tau]$ of type pr that respect $\Gamma$, and for all stores $\Sigma$,*

$$\langle\!\langle C[M] \,;\, \Sigma \rangle\!\rangle \approx_{sim} \langle\!\langle C[N] \,;\, \Sigma \rangle\!\rangle$$

Contextual equivalence is, by the form of its definition, a *congruence* – that is, it is preserved by all term constructors of the language. Furthermore, it is easy to show (using the confluence property for PCF), that if $M =_{pcf} N$, then $M \cong_{\forall C} N$. Hence the structure obtained by taking equivalence classes of open terms[3] under $\cong_{\forall C}$ is a model of typed lambda-calculus. This is the operationally-based term model we will use. Because this is a model, ordinary reasoning using $\beta$-conversion is valid in the model.

Definition 3.2 might seem unduly restrictive, because it requires terms to behave identically not only in every program context, but in every store. For terms that contain no location constants, however, the store adds no discriminative power:

**Lemma 3.3** *If $\Gamma \vdash M, N{:}\tau$, and $M$ and $N$ contain no location constants, then $\Gamma \vdash M \cong_{\forall C} N{:}\tau$ iff for all closing contexts $C[-_\tau]$ of type pr that respect $\Gamma$,*

$$\langle\!\langle C[M] \,;\, \emptyset \rangle\!\rangle \approx_{sim} \langle\!\langle C[N] \,;\, \emptyset \rangle\!\rangle$$

PROOF SKETCH: The forward direction is immediate. For the reverse direction, given a context $C$ and store $\Sigma$, it easy to construct a context $C'$ that builds $\Sigma$ and then executes $M$ in context $C$.

Our plan is to specify the semantics of our source language in this model by giving a compositional translation $[\![-]\!]$ from the source language to terms of the metalanguage; the meaning of a source language expression is the $\cong_{\forall C}$-equivalence class of the translated term. To show two source language expressions $e_1$ and $e_2$ have the same meaning, we must show that their translations lie in the same equivalence class, i.e., that $[\![e_1]\!] \cong_{\forall C} [\![e_2]\!]$.

## 4 Extensional Equivalence

The quantification over all contexts makes $\cong_{\forall C}$ difficult to establish directly. We must consider not only contexts in which the terms are executed, but also contexts that pass the terms around, put many copies of them in the store, etc.

Our major tool for easing this proof burden is a "context lemma" which states that two terms are contextually equivalent if and only if they are equivalent in a smaller set of contexts we call *extensional contexts.*

Extensional contexts are motivated by the observation that for each type constructor $(\to, +, \times)$, there is a natural way of determining whether two terms of such a type "behave the same way." At a function type $\alpha \to \beta$, two terms are extensionally equal if they take any term

[2] *respecting* $\Gamma$ means that the context binds the free variables of $M$ with binders of the correct type.

[3] Actually, terms open with respect to an extended type assignment $\Gamma^*$ [Gun92, p. 55].

$$\frac{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{write}\, n\ M'\ ;\ \Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \xrightarrow{!n}_{io} \langle\!\langle M'\ ;\ \Sigma'\rangle\!\rangle} \qquad \frac{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{read}\ M'\ ;\ \Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \xrightarrow{?n}_{io} \langle\!\langle M'\ n\ ;\ \Sigma'\rangle\!\rangle}$$

$$\frac{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{stop}\ ;\ \Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \xrightarrow{\surd}_{io} \langle\!\langle \mathsf{stopped}\ ;\ \Sigma'\rangle\!\rangle}$$

Figure 3: Semantics for I/O Operators

of type $\alpha$ to equivalent terms of type $\beta$. For product types, both projections should be equivalent, and for sum types, the case operator should produce equivalent terms. We can view these type-based contexts (application, projection, and case) as *type deconstructors*, as they decompose complex types into terms of their component types. If our only complex types were function types, then the extensional contexts would be exactly the head contexts of Milner's context lemma.

Figure 4 gives the grammar for extensional contexts. Each extensional context $T[-_\tau]{:}o$ is a context with a single hole of type $\tau$ that produces base type $o$.

We will say that two closed terms are extensionally equivalent iff the terms have equivalent base-type behavior in every extensional context. Open terms will be extensionally equivalent if all their substitution instances are. Equivalence for closed terms at base types, denoted $\cong_o$, is defined simply: Equivalent terms of type $pr$ must be in $\approx_{sim}$ for all stores $\Sigma$. For base types other than $pr$, terms must PCF-reduce to the same constant. For base type $\mathit{ref}(\tau)$ this means the same location constant. We omit the formal definition in this abstract. In the general case, the formal definition becomes:

**Definition 4.1 (extensional equivalence, $\cong_{ext}$)** *If $\Gamma \vdash M, N{:}\tau$, then $\Gamma \vdash M \cong_{ext} N{:}\tau$ iff for all closing substitutions $\sigma$ which respect $\Gamma$, and for all extensional contexts $T[-_\tau]{:}o$,*

$$T[M\sigma] \cong_o T[N\sigma]$$

This definition is similar to Gordon's notion of equivalence [Gor95]. His bisimilarity condition requires terms to be equivalent in exactly our extensional contexts; our base types correspond to his active types.

The main result of this section is that extensional and contextual equivalence coincide:

**Theorem 4.2 (context lemma)**

$$\Gamma \vdash M \cong_{ext} N{:}\tau \Leftrightarrow \Gamma \vdash M \cong_{\forall C} N{:}\tau$$

Proof sketch: First, we show that when extensionally equivalent closed terms are substituted into a term with a single free variable, the resulting terms are extensionally equivalent. This is the most complex portion of the proof, as it involves the inductive definition of substitution, the coinductive definition of $\approx_{sim}$, and the inductive definition of $\rightarrow^*_{sto}$. It is then easy to show that $\cong_{ext}$ is a congruence, and hence that $\cong_{ext} \subseteq \cong_{\forall C}$. The reverse inclusion follows by a standard argument [TW96]. □

Extensional equivalence is not the same as using logical relations; it imposes a lesser proof burden than methods based on logical relations, and also extends to untyped or recursively typed languages. Extensional equivalence requires only that terms behave equivalently in the *same* context. This contrasts with the logical-relations approach, in which terms are required to behave equivalently in all *equivalent* contexts. This in turn leads to non-monotonicity which is remedied only by restriction to finite types. That extensionally-equivalent terms do behave equivalently in equivalent contexts follows from Theorem 4.2.

## 5 Meyer-Sieber Examples

We now leave the metalanguage and turn to applications. We begin by considering the examples from [MS88]. Our techniques suffice to do all these examples; we give an illustrative one here.

Example 4 from [MS88] says:

*The block below always diverges.*

**begin**
  **new** $x$; **new** $y$;
  **procedure** *Twice*; **begin** $y := 2 * \mathit{contents}(y)$ **end**;
  $x := 0$; $y := 0$;
  $Q(\mathit{Twice})$;   % *Q is declared elsewhere*
  **if** $\mathit{contents}(x) = 0$ **then** *diverge* **fi**
**end**

In a language with I/O, the above block is *not* necessarily equivalent to *diverge*, as the procedure $Q$ might do I/O. So we will undertake to prove the equivalence of the following two blocks:

**begin**
  **new** $x$; **new** $y$;
  **procedure** *Twice*; **begin** $y := 2 * \mathit{contents}(y)$ **end**;
  $x := 0$; $y := 0$;
  $Q(\mathit{Twice})$;   % *Q is declared elsewhere*
  **if** $\mathit{contents}(x) = 0$ **then** *skip* **else** *diverge* **fi**
**end**

and

$$
\begin{array}{lll}
T[-_o]{:}o & ::= [-] & \text{base types} \\
T[-_{\tau_1\to\tau_2}]{:}o & ::= T[([-]\ M)]{:}o & \text{for all closed } M{:}\tau_1 \\
T[-_{\tau_1\times\tau_2}]{:}o & ::= T[\pi_1[-]]{:}o \mid T[\pi_2[-]]{:}o & \\
T[-_{\tau_1+\tau_2}]{:}o & ::= T[\mathsf{case}^{\gamma}_{\tau_1,\tau_2}[-]\ M_1\ M_2]{:}o & \text{for all } \gamma,\ M_1{:}\tau_1\to\gamma,\ M_2{:}\tau_2\to\gamma
\end{array}
$$

Figure 4: Grammar for Extensional Contexts

```
begin
  new x; new y;
  procedure Twice; begin y := 2 * contents(y) end;
  x := 0; y := 0;
  Q(Twice);                  % Q is declared elsewhere
end
```

where *skip* does nothing.

Our general plan for proving the equivalence of two source-program expressions, $e_1$ and $e_2$, is to specify a compositional translation $[\![-]\!]$ from the source language into our metalanguage, and then to show that $[\![e_1]\!] \cong_{ext} [\![e_2]\!]$.

The semantics of this Algol fragment is given by a straightforward CPS translation [App92, Plo75] into our metalanguage, where all blocks get translated to terms of type $pr{\to}pr$. After some simplification, the translation yields the following metalanguage term for the left-hand side:

$$
\begin{aligned}
[\![lhs]\!] = (\lambda\kappa{:}pr.\,&\mathsf{new}\,0\ (\lambda x{:}ref(int).\,\mathsf{new}\,0\ (\lambda y{:}ref(int).\\
&Q\ \mathit{Twice}\ (\mathsf{deref}\,x\ (\lambda v_x.\,zero?\ v_x\ \kappa\ \Omega_{pr})))))
\end{aligned}
$$

where *Twice* is defined as:

$$
\mathit{Twice} \stackrel{\text{def}}{=} (\lambda\kappa'{:}pr.\,\mathsf{deref}\,y\ (\lambda v_y.\,\mathsf{update}\,y\ (*\ 2\ v_y)\ \kappa'))
$$

and where $\Omega_\tau$ is the canonical divergent term $\mathsf{Y}(\lambda x{:}\tau.x)$.

The $[\![lhs]\!]$ term takes a run-time continuation $\kappa$, allocates fresh locations for $x$ and $y$ initialized to 0, and then continues with $x$ and $y$ bound to the fresh locations. The body of the block invokes the unknown function $Q$ with the closure *Twice* as its argument and with a continuation that checks if $x$ is zero and continues with $\kappa$ if so or diverges if not. The entire block has type $pr{\to}pr$.

The translation of the right-hand side, $[\![rhs]\!]$, is exactly the same as the left-hand side, except that the $(\mathsf{deref}\,x \ldots)$ continuation is replaced by simply $\kappa$.

To prove that the two translations are operationally equivalent, we appeal to the context lemma and set out to prove that for all terms $K$ of type $pr{\to}pr$ and all stores $\Sigma$,

$$
\langle\!\langle [\![lhs]\!]\,K\ ;\ \Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle [\![rhs]\!]\,K\ ;\ \Sigma\rangle\!\rangle
$$

The left-hand side will take a number of steps to the following state:

$$
\begin{aligned}
&\langle\!\langle (Q\ \mathit{Twice}\,(\mathsf{deref}\,x\ (\lambda v_x.\,zero?\ v_x\ K\ \Omega_{pr})))[l_x/x, l_y/y]\ ;\\
&\quad \Sigma[l_x \mapsto 0, l_y \mapsto 0]\rangle\!\rangle \qquad (l_x, l_y\ \mathit{fresh})
\end{aligned}
$$

Similarly, the right-hand side goes to:

$$
\begin{aligned}
&\langle\!\langle (Q\ \mathit{Twice}\,K)[l_x/x, l_y/y]\ ;\\
&\quad \Sigma[l_x \mapsto 0, l_y \mapsto 0]\rangle\!\rangle
\end{aligned}
$$

up to $\alpha_{loc}$.

We now construct a relation $S$, which includes the two states above, and we prove that $S$ is a bisimulation.

$$
\begin{aligned}
S \stackrel{\text{def}}{=} \{\ \ &(\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle\sigma_1,\ \langle\!\langle M\ ;\ \Sigma\rangle\!\rangle\sigma_2) \mid\\
&fv(M, rng(\Sigma)) \subseteq \{z\},\\
&\sigma_1 = [(\mathsf{deref}\,l_x\ (\lambda v_x.\,zero?\ v_x\ K\ \Omega_{pr}))/z],\\
&\sigma_2 = [K/z],\\
&\Sigma(l_x) = 0,\\
&l_x \notin locs(M, K, rng(\Sigma))\ \}
\end{aligned}
$$

This simulation $S$ characterizes the relevant aspects of the example – namely that $x$'s location $l_x$ is never updated (or, more exactly, that the only references to $l_x$ are within a term of the form $(\mathsf{deref}\,l_x \ldots)$).

The pair of states in which we are interested is clearly in $S$, with $M = (Q\ \mathit{Twice}\ z)$. Because $l_x$ is allocated fresh, we know that $Q$ and $K$ contain no references to $l_x$. The proof that $S$ is a bisimulation is straightforward.

This example illustrates both the power of this technique to prove equivalences but also its limitations: our translation does not validate the original equivalence from [MS88]; hence the translation is not fully abstract. Since, however, the languages that we are primarily interested in, like Scheme and ML, do have input-output behavior and do not obey stack discipline, the failure of full abstraction for Idealized Algol is not so important for our intended applications.

## 6 Assignment Elimination

For a more substantial example, we consider the problem of assignment elimination in Scheme. While Scheme has a rich set of expressed values, its variables denote only locations, because a Scheme program can always mutate a variable by performing a set! on it. The semantics of Scheme accomplishes this by automatically allocating storage for each parameter at every procedure call, and by automatically dereferencing each variable reference [RC+86].

Many Scheme compilers perform a preliminary pass called assignment elimination, in which Scheme is translated into an intermediate language in which the denoted values include expressed values as well, and in

$$
\begin{array}{rll}
e ::= & x \mid n \mid (\mathsf{lambda}\ (x \ldots)\ e) \mid (e\ e \ldots) & \text{expressions}\\
\mid & (\mathsf{set!}\ x\ e) \mid (\mathsf{letrec}\ ((x\ e) \ldots)\ e) & \\
p ::= & \mathsf{run}\ e & \text{programs}
\end{array}
$$

where $x$ ranges over identifiers and $n$ ranges over integers.

Figure 5: A subset of Scheme

$$
\begin{aligned}
&[\![n]\!]_{scm} = \lambda\kappa\colon Val\to pr.\ in_N\ n\ \kappa\\
&[\![x]\!]_{scm} = \lambda\kappa\colon Val\to pr.\ \mathsf{deref}\, x\ \kappa\\
&[\![(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)]\!]_{scm} = \lambda\kappa\colon Val\to pr.\\
&\qquad in_F\,(\lambda v^*\colon Val^*.\lambda k'\colon Val\to pr.\\
&\qquad\quad \mathsf{new}^{Val}(listref_1\ v^*)\ (\lambda x_1{:}ref(Val).\cdots\\
&\qquad\qquad \mathsf{new}^{Val}(listref_n\ v^*)\ (\lambda x_n{:}ref(Val).[\![e]\!]_{scm}\ \kappa')\ldots))\ \kappa\\
&[\![(e_0\ e_1 \ldots e_n)]\!]_{scm} = \lambda\kappa\colon Val\to pr.[\![e_0]\!]_{scm}(\lambda f\colon Val.\\
&\qquad [\![e_1]\!]_{scm}(\lambda v_1\colon Val.\cdots\\
&\qquad\quad [\![e_n]\!]_{scm}(\lambda v_n\colon Val.\\
&\qquad\qquad out_F\ f\ (\lambda f'\colon Val^*\to(Val\to pr)\to pr.\\
&\qquad\qquad\quad f'\ \langle v1,\ldots,v_n\rangle\ \kappa))\ldots))\\
&[\![(\mathsf{set!}\ x\ e)]\!]_{scm} = \lambda\kappa\colon Val\to pr.[\![e]\!]_{scm}(\lambda v\colon Val.\ \mathsf{update}\, x\ v\ (\kappa\ \Omega_{Val}))\\
&[\![(\mathsf{letrec}\ ((x_1\ e_1)\ldots(x_n\ e_n))\ e)]\!]_{scm} = \lambda\kappa\colon Val\to pr.\\
&\qquad \mathsf{new}\,\Omega_{Val}(\lambda x_1{:}ref(Val).\cdots \mathsf{new}\,\Omega_{Val}(\lambda x_n{:}ref(Val).\\
&\qquad\quad [\![e_1]\!]_{scm}(\lambda v_1\colon Val.\ \mathsf{update}\, x_1 v_1 \cdots\\
&\qquad\qquad [\![e_n]\!]_{scm}(\lambda v_n\colon Val.\ \mathsf{update}\, x_n v_n\ ([\![e]\!]_{scm}\kappa)\cdots)\\
&[\![\mathsf{run}\ e]\!]_{scm} = \langle\!\langle C_0[[\![e]\!]_{scm}]\ ;\ \emptyset\rangle\!\rangle
\end{aligned}
$$

Figure 6: CPS translation from Scheme subset to metalanguage

which allocation and dereferencing are explicit. Variables that are demonstrably never mutated are bound directly to expressed values, and those that may be mutated are bound to explicitly-allocated cells [KKsR+86].

Although this translation is generally considered unproblematic, its correctness is difficult to prove, because it radically changes the store behavior of the program. Here we sketch the first known proof of correctness for this transformation. We first describe the source and target language of the transformation. We formulate the translation, and then we sketch the proof itself.

## 6.1 Source Language Semantics

We will consider the subset of Scheme given by the grammar in Figure 5. The language consists of expressions $e$ and programs $p$.

We specify the semantics of this Scheme subset by a straightforward CPS translation (Figure 6). The main difference from an ordinary denotational semantics is that it does not introduce an environment; instead, it leaves free variables in the source term as free variables in the translation.

The type $Val$ of expressed values is given as a recursive sum:

$$
\begin{array}{ll}
Val = N + F & \text{expressed values}\\
N = int & \text{integers}\\
F = Val^*\to(Val\to pr)\to pr & \text{procedures}
\end{array}
$$

The denoted values are of type $ref(Val)$. The translation of any expression is a term of type $(Val\to pr)\to pr$, given that all the free variables have type $ref(Val)$.

For each summand $S$ of the $Val$ type, we use CPS injection and extraction combinators, $in_S : S\to(Val\to pr)\to pr$ and $out_S : Val\to(S\to pr)\to pr$. We also use $\langle -,\ldots,-\rangle$ and $listref_i$ for the evident list operations. All these can be easily defined in the metalanguage.

The rule for an abstraction $(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)$ injects into the $Val$ type a closure. When the closure is invoked, storage is allocated for each formal, the cells are initialized, and the body is executed in an environment in which the formals are bound to the new locations.

The rule for assignment, $(\mathsf{set!}\ x\, e)$, updates the location bound to $x$ with the value of $[\![e]\!]_{scm}$. The Scheme semantics calls for the continuation to be invoked with an undefined value, for which we use $\Omega_{Val}$, the canonical nonterminating term of type $Val$.

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid (\mathsf{lambda}\ (x \ldots)\ e) \mid (e\ e \ldots) \\
& \mid & (\mathsf{letrec}\ ((x\ e) \ldots)\ e) \\
& \mid & (\mathsf{let}\ ((x\ \mathit{rhs}) \ldots)\ e \\
& \mid & (\mathsf{cell\text{-}ref}\ x) \mid (\mathsf{cell\text{-}set!}\ x\ e) \\
\mathit{rhs} & ::= & x \mid (\mathsf{make\text{-}cell}\ e)
\end{array}
$$

Figure 7: Syntax of $\text{Scheme}_{cell}$

$$[\![x]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.\kappa\, x$$

$$
\begin{aligned}
[\![(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)]\!]_{cell} = {} & \lambda\kappa{:}\,Val{\rightarrow}pr. \\
& in_F\,(\lambda v^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\rightarrow}pr. \\
& \quad [\![e]\!]_{cell}[(listref_1\, v^*)/x_1, \ldots, (listref_n\, v^*)/x_n]\ \kappa')\ \kappa
\end{aligned}
$$

$$[\![(\mathsf{make\text{-}cell}\ e)]\!]_{cell} = \lambda\kappa{:}ref(Val){\rightarrow}pr.[\![e]\!]_{cell}(\lambda v{:}\,Val.\ \mathsf{new}\ v\ \kappa)$$

$$[\![(\mathsf{cell\text{-}ref}\ x)]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.\ \mathsf{deref}\ x\ \kappa$$

$$[\![(\mathsf{cell\text{-}set!}\ \ x\ e)]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.[\![e]\!]_{cell}(\lambda v{:}\,Val.\ \mathsf{update}\ x\ v\ (\kappa\,\Omega)))$$

Figure 8: CPS translation from $\text{Scheme}_{cell}$ to metalanguage

The rule for recursive definitions follows the Scheme semantics [RC+86]. It first allocates all required locations, initializing them to a dummy value, and then updates all of the locations with their contents within a scope that includes bindings from all mutually recursive identifiers $x_1, \ldots, x_n$ to their newly allocated locations. Because Scheme allows arbitrary side-effects in the $e_i$, the usual definition with the Y operator is impossible.

Finally, evaluation of a program runs the expression in a suitable initial context and an empty store. The context $C_0[-]$ contains a preamble that loads the values of the primitives of Scheme into locations in the store, binds identifiers to those locations, and then runs the program in an initial continuation (probably something like $(\lambda v{:}\,Val.\,print_{Val}\,v\ \mathsf{stop})$ for some suitable metalanguage term $print_{Val}$). Our definition is flexible enough to accommodate the possibility of primitives that perform input-output or do non-local jumps, like call/cc.

This semantics easily verifies the equivalence (1) with which we began this paper.

## 6.2 $\text{Scheme}_{cell}$

Figure 7 gives the syntax of the intermediate language $\text{Scheme}_{cell}$, which includes new cell operations and no longer contains set!. Figure 8 gives the translation from $\text{Scheme}_{cell}$ into our metalanguage. Expressed values are still of type *Val*; identifiers denote either values of type *Val* or cells of type $ref(Val)$. It is easy to deduce statically which variables are which; we omit the easy type-checking system.

The rule for variables no longer automatically does a dereference, and the rule for abstraction no longer allocates storage automatically. Instead, cell allocation and dereferencing are moved into the cell operations. The rule for letrec is identical to that for $[\![-]\!]_{scm}$. The rule for let does the evident thing and is omitted.

## 6.3 The Translation

Our goal is to translate our Scheme subset to $\text{Scheme}_{cell}$, using cells only for variables that are potentially mutated. Figure 9 gives the translation. Bound variables that do not appear on the left-hand side of a set! expression are translated as is, so no storage is allocated for them; all other variables are translated by explicitly allocating a location at binding time, and by explicitly dereferencing when the variable is used. Variables that are bound by a letrec are initialized by mutation, and are therefore treated as cells.

## 6.4 Correctness of Assignment Elimination

Our goal is to show that assignment elimination is meaning-preserving, that is:

**Theorem 6.2 (Correctness of Assignment Elimination)** *For any Scheme expression $e$, and any $\Gamma$ mapping the free variables of $[\![e]\!]_{scm}$ and $[\![AE(e)]\!]_{cell}$ to $ref(Val)$,*

$$\Gamma \vdash [\![e]\!]_{scm} \cong_{\forall C} [\![AE(e)]\!]_{cell}{:}(Val{\rightarrow}pr){\rightarrow}pr$$

PROOF:

**Definition 6.1 (Assignment Elimination ($AE$))** *Let $sv(e)$ be the set of free variables of e that appear in e as the left-hand side of a* set! *expression. Define $AE(e)$ to be $AE(\emptyset, e)$, where $AE(V,e)$ is defined as follows. Here V is the set of free variables that are to be bound to values rather than locations.*

$$
\begin{array}{l}
AE(V,x) = \begin{cases} x & \text{if } x \in V \\ (\textsf{cell-ref}\ x) & \text{if } x \notin V \end{cases} \\
AE(V, (\textsf{set!}\ x\ e)) = (\textsf{cell-set!}\ x\ AE(V,e)) \\
AE(V, (e_0\, e_1\ \ldots\ e_n)) = (AE(V,e_0)\ \ AE(V,e_1)\ \ldots\ AE(V,e_n)) \\
AE(V, (\textsf{lambda}\ (x_1\ \ldots\ x_n)\ e)) \\
\quad = (\textsf{lambda}\ (x_1\ \ldots\ x_n)\ (\textsf{let}\ ((x_1\ r_1)\ \ldots\ (x_n\ r_n))\ AE(V',e))) \\
\qquad \text{where} \begin{cases} r_i = \begin{cases} (\textsf{make-cell}\ x_i) & \text{if } x_i \in sv(e) \\ x_i & \text{if } x_i \notin sv(e) \end{cases} \\ V' = V \cup \{x_i \mid x_i \notin sv(e)\} - \{x_i \mid x_i \in sv(e)\} \end{cases} \\
AE(V, (\textsf{letrec}\ ((x_1\ e_1) \ldots (x_n\ e_n))\ e)) \\
\quad = (\textsf{letrec}\ ((x_1\ \ AE(V',e_1)) \ldots (x_n\ \ AE(V',e_n))) \\
\qquad AE(V',e)) \qquad\qquad\qquad \text{where } V' = V - \{x_1, \ldots, x_n\}
\end{array}
$$

Figure 9: Assignment Elimination

1. We first need to check that
$$\Gamma \vdash [\![AE(e)]\!]_{cell} : (Val \rightarrow pr) \rightarrow pr$$
This is straightforward.

2. Hence, by the Context Lemma (Theorem 4.2) and the definition of $\cong_{ext}$, it suffices to show that for all closing substitutions $\sigma$ mapping free variables to closed terms of type $ref(Val)$, all closed terms $K$ of type $Val \rightarrow pr$, and all stores $\Sigma$,
$$\langle\!\langle ([\![e]\!]_{scm}\sigma)\ K\ ;\ \Sigma \rangle\!\rangle \approx_{sim} \langle\!\langle ([\![AE(e)]\!]_{cell}\sigma)\ K\ ;\ \Sigma \rangle\!\rangle$$

3. To show these states are bisimilar, we construct a relation $S_{AE}$ between metalanguage states that characterizes the syntactic differences between (non-transformed) Scheme programs and those programs after assignment elimination, as they reduce under $\xrightarrow{\alpha}_{io}$. This construction involves several stages.

4. As the original program and its AE-translated version run, there are several different ways in which corresponding subterms can differ:

   (a) The translations of Scheme lambda expressions will differ in the bodies of their closures. The left-hand side closure body will contain new operators for all local parameters, whereas the right-hand side will only allocate storage for mutable local parameters. On the left-hand side, references to local variables will always involve store lookup with the deref operator, whereas on the right-hand side, references to immutable variables will not do store lookup, as the variable will get its value via substitution. The macros $\mathcal{L}_1$ and $\mathcal{R}_1$ in Figure 10 capture the different translations of a Scheme lambda expression. Note that the two equations at the top of the Figure 10 are not definitions; they are instead specifications for the term macros $\mathcal{L}_1$ and $\mathcal{R}_1$. The figure then gives definitions for $\mathcal{L}_1$ and $\mathcal{R}_1$ that satisfy these specifications. The same pattern is followed in the remaining four blocks of Figure 10.

   (b) When subterms corresponding to lambda expressions are applied to a run-time continuation, they are injected into the $F$ summand of the $Val$ type and sent to the continuation. The macros $\mathcal{L}_2$ and $\mathcal{R}_2$ in Figure 10 express the form of the two sides after they are applied to run-time continuation terms.

   (c) When the closures are used at run-time (i.e., applied to arguments), they are extracted from the $Val$ type and applied to their application-time argument list and continuation. At this point, the differences between the closure bodies are "exposed", and we see different storage behavior. This stage of execution is expressed by the macros $\mathcal{L}_3$ and $\mathcal{R}_3$ in Figure 10.

   (d) As the exposed bodies of corresponding closures reduce, the left-hand side allocates storage for all local variables, whereas the right-hand side only allocates storage for mutable local variables. For immutable local variables, on the left-hand side we have a substitution $[l_i/x_i]$ for each $x_i$ immutable variable (and mutable too, for that matter). On the right-hand side, for the immutable variables $x_i$, we see value substitutions, $[(listref_i\ v^*)/x_i]$, where the variable $v^*$ contains the list of incoming arguments. We express this relation by first

Initial Translation:

$$[\![(\mathsf{lambda}\ (x_1 \ldots x_m)\ e)]\!]_{scm} = \mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![\ [\![e]\!]_{scm}\ ]\!]$$

$$\begin{aligned}\mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![P]\!] &\stackrel{\text{def}}{=} \\ &(\lambda\kappa{:}\mathit{Val}{\to}\mathit{pr}.\ \mathit{in}_F(\lambda v^*{:}\mathit{Val}^*.\lambda\kappa'{:}\mathit{Val}{\to}\mathit{pr}. \\ &\quad \mathsf{new}^{\mathit{Val}}(\mathit{listref}_1\ v^*)\ (\lambda x_1{:}\mathit{ref}(\mathit{Val}).\cdots \\ &\quad\quad \mathsf{new}^{\mathit{Val}}(\mathit{listref}_m\ v^*)\ (\lambda x_m{:}\mathit{ref}(\mathit{Val}). \\ &\quad\quad\quad (P\ \kappa')\ldots))))\ \kappa)\end{aligned}$$

$$\begin{aligned}&[\![AE(V,(\mathsf{lambda}\ (x_1 \ldots x_m)\ e))]\!]_{cell} = \\ &\quad \mathcal{R}_1(V',C)[\![\ [\![AE(V',e)]\!]_{cell}\ ]\!],\ \text{with} \\ &\quad V' = V \cup \{x_i \mid x_i \notin sv(e)\} - \{x_i \mid x_i \in sv(e)\} \\ &\quad C = \{x_i \mid x_i \in sv(e)\}\end{aligned}$$

$$\begin{aligned}\mathcal{R}_1(V,C)[\![P]\!] &\stackrel{\text{def}}{=} \\ &(\lambda\kappa{:}\mathit{Val}{\to}\mathit{pr}.\ \mathit{in}_F(\lambda v^*{:}\mathit{Val}^*.\lambda\kappa'{:}\mathit{Val}{\to}\mathit{pr}. \\ &\quad \left.\begin{array}{l}\mathsf{new}^{\mathit{Val}}(\mathit{listref}_i\ v^*) \\ (\lambda x_i{:}\mathit{ref}(\mathit{Val}).\end{array}\right\}\ \text{for each } x_i \in C \\ &\quad ((P\ [(\mathit{listref}_i\ v^*)/x_i]_{x_i \in V})\ \kappa')\ldots))\ \kappa)\end{aligned}$$

Applied to a continuation $K$:

$$\begin{aligned}&\langle\!\langle(\mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![P]\!]\ K)\ ;\ \Sigma\rangle\!\rangle \to^*_{pcf} \\ &\quad \langle\!\langle(K\ \mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!])\ ;\ \Sigma\rangle\!\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!] &\stackrel{\text{def}}{=} \\ &\mathsf{inr}(\lambda v^*{:}\mathit{Val}^*.\lambda\kappa'{:}\mathit{Val}{\to}\mathit{pr}. \\ &\quad \mathsf{new}^{\mathit{Val}}(\mathit{listref}_1\ v^*)\ (\lambda x_1{:}\mathit{ref}(\mathit{Val}).\cdots \\ &\quad\quad \mathsf{new}^{\mathit{Val}}(\mathit{listref}_m\ v^*)\ (\lambda x_m{:}\mathit{ref}(\mathit{Val}). \\ &\quad\quad\quad (P\ \kappa')\ldots)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle(\mathcal{R}_1(V,C)[\![P]\!]\ K)\ ;\ \Sigma\rangle\!\rangle \to^*_{pcf} \\ &\quad \langle\!\langle(K\ \mathcal{R}_2(V,C)[\![P]\!])\ ;\ \Sigma\rangle\!\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{R}_2(V,C)[\![P]\!] &\stackrel{\text{def}}{=} \\ &\mathsf{inr}(\lambda v^*{:}\mathit{Val}^*.\lambda\kappa'{:}\mathit{Val}{\to}\mathit{pr}. \\ &\quad \left.\begin{array}{l}\mathsf{new}^{\mathit{Val}}(\mathit{listref}_i\ v^*) \\ (\lambda x_i{:}\mathit{ref}(\mathit{Val}).\end{array}\right\}\ \text{for each } x_i \in C \\ &\quad ((P\ [(\mathit{listref}_i\ v^*)/x_i]_{x_i \in V})\ \kappa')\ldots))\end{aligned}$$

Closure Applied to an Argument List $Q$:

$$\begin{aligned}&\langle\!\langle\ \mathit{out}_F(\mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!])\ (\lambda f{:}\mathit{Val}^*{\to}(\mathit{Val}{\to}\mathit{pr}){\to}\mathit{pr}. \\ &\quad f\ Q\ K)\ ;\ \Sigma\rangle\!\rangle \\ &\ \ \text{where } Q{:}\mathit{Val}^* \text{ and } K{:}\mathit{Val}{\to}\mathit{pr} \\ &\ \ \to^*_{pcf}\ \langle\!\langle\mathcal{L}_3\langle x_1 \ldots x_m\rangle[\![P\ Q\ K]\!]\ ;\ \Sigma\rangle\!\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{L}_3\langle x_1 \ldots x_m\rangle[\![P_1\ P_2\ P_3]\!] &\stackrel{\text{def}}{=} \\ &\mathsf{new}^{\mathit{Val}}(\mathit{listref}_1\ P_2)\ (\lambda x_1{:}\mathit{ref}(\mathit{Val}).\cdots \\ &\quad \mathsf{new}^{\mathit{Val}}(\mathit{listref}_m\ P_2)\ (\lambda x_m{:}\mathit{ref}(\mathit{Val}). \\ &\quad\quad (P_1\ P_3)\ldots)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle\ \mathit{out}_F(\mathcal{R}_2(V,C)[\![P]\!])\ (\lambda f{:}\mathit{Val}^*{\to}(\mathit{Val}{\to}\mathit{pr}){\to}\mathit{pr}. \\ &\quad f\ Q\ K)\ ;\ \Sigma\rangle\!\rangle \\ &\ \ \text{where } Q{:}\mathit{Val}^* \text{ and } K{:}\mathit{Val}{\to}\mathit{pr} \\ &\ \ \to^*_{pcf}\ \langle\!\langle\mathcal{R}_3(V,C)[\![P\ Q\ K]\!]\ ;\ \Sigma\rangle\!\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{R}_3(V,C)[\![P_1\ P_2\ P_3]\!] &\stackrel{\text{def}}{=} \\ &\quad \left.\begin{array}{l}\mathsf{new}^{\mathit{Val}}(\mathit{listref}_i\ P_2) \\ (\lambda x_i{:}\mathit{ref}(\mathit{Val}).\end{array}\right\}\ \text{for each } x_i \in C \\ &\quad ((P_1\ [(\mathit{listref}_i\ P_2)/x_i]_{x_i \in V})\ P_3)\ldots)\end{aligned}$$

Figure 10: Definitions of $\mathcal{L}_i$, $\mathcal{R}_i$ macros

constructing a relation $\stackrel{AE}{\leadsto}_V$ between terms that captures the differences between translations of immutable variable references. This is done in Figure 11. The subscript $V$ is a list of variables which are treated as values on the right-hand side. We then construct a relation $\stackrel{AE}{\leadsto}_{V|Z}$ between metalanguage terms, defined in Figure 12, that characterizes the differences between terms which were originally related by $\stackrel{AE}{\leadsto}_V$ but which have been subjected to run-time substitutions – of locations for variables on the left-hand side and of values for variables on the right-hand side. We then extend $\stackrel{AE}{\leadsto}_{V|Z}$ to act on stores pointwise.

5. It is easy to show that if $V = \{x_1, \ldots, x_n\}$ and $\Gamma[x_i{:}\mathit{ref}(\mathit{Val})]_{i=1}^n \vdash M{:}\tau$ and

$$\Gamma[x_i{:}\mathit{ref}(\mathit{Val})]_{i=1}^n \vdash M \stackrel{AE}{\leadsto}_V N{:}\tau$$

then $\Gamma[x_i{:}\mathit{Val}]_{i=1}^n \vdash N{:}\tau$, and that if $\Gamma[x{:}\tau'] \vdash M \stackrel{AE}{\leadsto}_V N{:}\tau$ and $\Gamma \vdash P \stackrel{AE}{\leadsto}_V P'{:}\tau'$ and $x \notin V$, then $\Gamma \vdash M[P/x] \stackrel{AE}{\leadsto}_V N[P'/x]{:}\tau$.

6. Figure 13 defines the candidate relation, $S_{AE}$, by abstracting over corresponding pairs of terms which differ as outlined earlier. Each of the distinct ways in which corresponding metalanguage subterms might differ is handled by a substitution pair $(\sigma^i_{\mathcal{L}}, \sigma^i_{\mathcal{R}})$. Each substitution has a domain $u^i_1, \ldots, u^i_{n_i}$, which mark the locations of the corresponding subterms that differ as described in step 4(a)-(c) above.

$$\Gamma \vdash n \overset{AE}{\leadsto}_V n{:}int \quad (\textsc{cong-int})$$

$$\Gamma \vdash c \overset{AE}{\leadsto}_V c{:}\tau \qquad (\tau \text{ from Figure 1}) \quad (\textsc{cong-const})$$

$$\Gamma \vdash l^\tau \overset{AE}{\leadsto}_V l^\tau{:}ref(\tau) \quad (\textsc{cong-loc})$$

$$\frac{\Gamma[x{:}\tau'] \vdash M \overset{AE}{\leadsto}_V N{:}\tau \quad x \notin V}{\Gamma \vdash (\lambda x{:}\tau'.M) \overset{AE}{\leadsto}_V (\lambda x{:}\tau'.N){:}\tau' \to \tau} \quad (\textsc{cong-abs})$$

$$\frac{\Gamma \vdash M \overset{AE}{\leadsto}_V M'{:}\tau' \to \tau \quad \Gamma \vdash N \overset{AE}{\leadsto}_V N'{:}\tau'}{\Gamma \vdash (M\ N) \overset{AE}{\leadsto}_V (M'\ N'){:}\tau} \quad (\textsc{cong-app})$$

$$\frac{x \notin V \quad \Gamma(x) = \tau}{\Gamma \vdash x \overset{AE}{\leadsto}_V x{:}\tau} \quad (\textsc{mutable-var})$$

$$\frac{\Gamma \vdash M \overset{AE}{\leadsto}_V N{:}\,Val \to pr \quad x \in V \quad \Gamma(x) = ref(Val)}{\Gamma \vdash (\mathsf{deref}\, x\ M) \overset{AE}{\leadsto}_V (N\ x){:}pr} \quad (\textsc{immutable-var})$$

Figure 11: Definition of $\overset{AE}{\leadsto}_V$

The last section of the definition of $S_{AE}$ requires all of the closure bodies to be $\overset{AE}{\leadsto}_{V|Z}$-related. In order to simplify the notation, we assume that all of the bound variables in the states we consider are distinct. This allows us to have a single $\overset{AE}{\leadsto}_{V|Z}$ relation for the whole simulation relation, rather than different $V$'s and $Z$'s for different closures.

7. For any Scheme expression $e$, any metalanguage context $C[-_{(Val \to pr) \to pr}]{:}pr$, and any store $\Sigma$,

$$\langle\!\langle C[\llbracket e \rrbracket_{scm}]\ ;\ \Sigma \rangle\!\rangle\ S_{AE}\ \langle\!\langle C[\llbracket AE(e) \rrbracket_{cell}]\ ;\ \Sigma \rangle\!\rangle$$

To establish this, let the $Z$ in the $\overset{AE}{\leadsto}_{V|Z}$ relation be $\langle\rangle$ (the empty sequence), let the $\sigma^2$ and $\sigma^3$ substitutions both be empty, let the $\sigma^1$ substitutions have elements for the translation of each lambda expression in $e$, and let the $V$ in $\overset{AE}{\leadsto}_V$ contain the immutable local variables of the lambda-expressions in $e$. The proof uses the results of step 5 to deal with binding and substitution.

By choosing $C$ appropriately, this implies that all the states in step 2 above will be in $S_{AE}$, as desired.

8. We must verify that $S_{AE}$ is a bisimulation relation, using the definition of a bisimulation. This in turn requires an induction on the number of steps to an input-output operation, which is done using a case analysis on the form of $M_0$ and $N_0$; the interesting cases are when these terms have one of the $u^i_j$ as a head variable. The most delicate step is to confirm that subterms of reference type reduce to the same location constant when they are in $\overset{AE}{\leadsto}_{V|Z}$: If $M \overset{AE}{\leadsto}_{V|Z} N$, $M, N{:}ref(\tau)$, and $(M, N) \notin Z$, then

$$M \to^*_{pcf} l \Leftrightarrow N \to^*_{pcf} l$$

The proof is by induction on the length of the reduction.

9. Thus $S_{AE}$ is a bisimulation relation that contains each of the required pairs of terms, and we are done.

## 7 Open Problems and Future Work

We believe that these techniques are powerful enough to treat other important optimizations, such as lambda-lifting in Scheme. We would like to consider more such transformations. In particular, we would like to consider the interaction between these techniques and those relying on flow analysis to collect global data about the program.

It is clear that the semantics of Idealized Algol in Section 5 is not fully abstract, because our contexts include procedures with input-output and non-local jumps. On the other hand, full abstraction for our Scheme subset is not a well-formulated question, since the initial context $C_0$ is unspecified. Our Scheme semantics is as abstract as possible, in that for any Scheme expressions $e_1$ and $e_2$, if $\llbracket e_1 \rrbracket_{scm} \not\cong_{\forall C} \llbracket e_2 \rrbracket_{scm}$, there is some choice of $C_0$ that distinguishes them. This remains a matter for further investigation.

**Definition 6.3** ($\overset{AE}{\rightsquigarrow}_{V|Z}$ **for metalanguage terms)** *For* $V = \langle x_1, \ldots, x_n \rangle$, $Z = \langle (l_1, P_1), \ldots, (l_k, P_k) \rangle$,

$$
\begin{array}{l}
\Gamma \vdash M \overset{AE}{\rightsquigarrow}_{V|Z} N{:}\tau \Leftrightarrow \\
\quad \wedge\ k \leq n \\
\quad \wedge\ \Gamma(x_i) = \mathit{ref}(\mathit{Val}), \quad \mathit{for}\ 1 \leq i \leq n \\
\quad \wedge\ P_i{:}\mathit{Val},\ \mathit{for}\ 1 \leq i \leq k \\
\quad \wedge\ \exists M',\ N'\ \text{s.t.} \\
\qquad \wedge\ M = M'[l_1/x_1, \ldots, l_k/x_k] \\
\qquad \wedge\ N = N'[P_1/x_1, \ldots, P_k/x_k] \\
\qquad \wedge\ \Gamma \vdash M' \overset{AE}{\rightsquigarrow}_{V} N'{:}\tau
\end{array}
$$

**Definition 6.4** ($\overset{AE}{\rightsquigarrow}_{V|Z}$ **for stores)**

$$
\begin{array}{l}
\Gamma \vdash \Sigma'_M \overset{AE}{\rightsquigarrow}_{V|Z} \Sigma_N \Leftrightarrow \\
\quad \wedge\ \mathit{dom}(\Sigma'_M) = \mathit{dom}(\Sigma_N) \\
\quad \wedge\ \forall \tau, l^\tau \in \mathit{dom}(\Sigma'_M)\ .\ \Gamma \vdash \Sigma'_M(l) \overset{AE}{\rightsquigarrow}_{V|Z} \Sigma_N(l){:}\tau
\end{array}
$$

Figure 12: The Relations $\overset{AE}{\rightsquigarrow}_{V|Z}$ between Terms and between Stores

$$
\begin{array}{l}
S_{AE} \overset{\text{def}}{=} \{ (\langle\!\langle M_0\ ;\ \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}, \langle\!\langle N_0\ ;\ \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}) \mid \\
\qquad \sigma_{\mathcal{L}} = \sigma^1_{\mathcal{L}} \cup \sigma^2_{\mathcal{L}} \cup \sigma^3_{\mathcal{L}} \\
\qquad \sigma_{\mathcal{R}} = \sigma^1_{\mathcal{R}} \cup \sigma^2_{\mathcal{R}} \cup \sigma^3_{\mathcal{R}} \\
\qquad \mathit{fv}(M, \mathit{rng}(\Sigma)) \subseteq \{u^1_1, \ldots, u^1_{n_1}, u^2_1, \ldots, u^2_{n_2}, u^3_1, \ldots, u^3_{n_3}\}
\end{array}
$$

$$
\begin{array}{c}
(\mathcal{L}_1, \mathcal{R}_1) \\ \text{pairs}
\end{array}
\left\{
\begin{array}{l}
\sigma^1_{\mathcal{L}} = [u^1_1 \mapsto \mathcal{L}_1(X^1_1)[\![M^1_1]\!], \ldots, u^1_{n_1} \mapsto \mathcal{L}_1(X^1_{n_1})[\![M^1_{n_1}]\!]] \\
\sigma^1_{\mathcal{R}} = [u^1_1 \mapsto \mathcal{R}_1(V^1_1, C^1_1)[\![N^1_1]\!], \ldots, u^1_{n_1} \mapsto \mathcal{R}_1(V^1_{n_1}, C^1_{n_1})[\![N^1_{n_1}]\!]] \\
n_1 = \#\sigma^1_{\mathcal{L}} = \#\sigma^1_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{c}
(\mathcal{L}_2, \mathcal{R}_2) \\ \text{pairs}
\end{array}
\left\{
\begin{array}{l}
\sigma^2_{\mathcal{L}} = [u^2_1 \mapsto \mathcal{L}_2(X^2_1)[\![M^2_1]\!], \ldots, u^2_{n_2} \mapsto \mathcal{L}_2(X^2_{n_2})[\![M^2_{n_2}]\!]] \\
\sigma^2_{\mathcal{R}} = [u^2_1 \mapsto \mathcal{R}_2(V^2_1, C^2_1)[\![N^2_1]\!], \ldots, u^2_{n_2} \mapsto \mathcal{R}_2(V^2_{n_2}, C^2_{n_2})[\![N^2_{n_2}]\!]] \\
n_2 = \#\sigma^2_{\mathcal{L}} = \#\sigma^2_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{c}
(\mathcal{L}_3, \mathcal{R}_3) \\ \text{pairs}
\end{array}
\left\{
\begin{array}{l}
\sigma^3_{\mathcal{L}} = [u^3_1 \mapsto \mathcal{L}_3(X^3_1)[\![M^3_{1,1}\ M^3_{1,2}\ M^3_{1,3}]\!], \ldots, u^3_{n_3} \mapsto \mathcal{L}_3(X^3_{n_3})[\![M^3_{n_3,1}\ M^3_{n_3,2}\ M^3_{n_3,3}]\!]] \\
\sigma^3_{\mathcal{R}} = [u^3_1 \mapsto \mathcal{R}_3(V^3_1, C^3_1)[\![N^3_{1,1}\ N^3_{1,2}\ N^3_{1,3}]\!], \ldots, u^3_{n_3} \mapsto \mathcal{R}_3(V^3_{n_3}, C^3_{n_3})[\![N^3_{n_3,1}\ N^3_{n_3,2}\ N^3_{n_3,3}]\!]] \\
n_3 = \#\sigma^3_{\mathcal{L}} = \#\sigma^3_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{l}
\quad \exists \Gamma, \Sigma'_M \wedge \\
\quad \exists V = \langle x_1, \ldots, x_n \rangle \wedge \\
\quad \exists Z = \langle (l_1, P_1), \ldots, (l_k, P_k) \rangle, k \leq n\ \text{s.t.} \\
\qquad \text{for each} (M, N) \in \{(M^i_j, N^i_j)\} \cup \{(M_0, N_0)\}, \\
\qquad \wedge\ \Gamma \vdash M \overset{AE}{\rightsquigarrow}_{V|Z} N{:}\tau \\
\qquad \wedge\ \Sigma_M = \Sigma'_M[l_1 \mapsto P'_1, \ldots, l_k \mapsto P'_k] \\
\qquad \wedge\ \Gamma \vdash \Sigma'_M \overset{AE}{\rightsquigarrow}_{V|Z} \Sigma_N \\
\qquad \wedge\ \Gamma \vdash P'_i \overset{AE}{\rightsquigarrow}_{V|Z} P_i, 1 \leq i \leq k \\
\quad \}
\end{array}
$$

Figure 13: Definition of $S_{AE}$

Our formalism should make it relatively easy to extend our input-output primitives to model communicating programs; how this will change the theoretical results remains to be seen.

## 8 Conclusions

We have given a denotational semantics for programs with input-output and a heap-allocated store. Because the framework is denotational, it encompasses a variety of source languages, and we need not reprove complex technical results like the Context Lemma each time we change the source language. Because equivalence in the model is based on input-output behavior, we can classify the behavior even of programs that do not terminate. (Space precludes an example here, see [TW96]). Similarly, formulating equivalence in terms of observable input-output behavior sidesteps most of the problems associated with store semantics.

Because equivalence in the meta-language is based on input-output behavior, proofs of equivalence ultimately rely on bisimulation arguments. In our experience, these arguments are intuitively satisfying, since they formalize the notion that the two programs being considered "stay in sync" as they compute. This approach imposes a lesser proof burden than do methods based on logical relations, and also extends to untyped or recursively typed languages.

We have used the technique to obtain a new result: the correctness of assignment elimination in Scheme. We believe that our methods will extend to a variety of transformations used in Scheme and ML compilers.

## References


[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[CG95] R. L. Crole and A. D. Gordon. A sound metalogical semantics for input/output effects. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer science logic: 8th workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 339–353, Berlin, Heidelberg, and New York, 1995. Springer-Verlag.

[CH94] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for scheme. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pages 128–139, 1994.

[Fel87] Matthias Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, Cambridge, 1994.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of 11th Conference on Mathematical Foundations of Programming Semantics*, 1995.

[GSR95] J. D. Guttman, V. Swarup, and J. Ramsdell. The VLISP verified scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, 1995.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, 1992.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conf. Rec. 16th ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[KKsR+86] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[Mil77] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mor68] James H. Morris, Jr. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, Cambridge, MA, 1968.

[MS88] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conf. Rec. 15th ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.

[MT91] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[Ode94] Martin Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 48–59, 1994.

[Ole85] F. J. Oles. Type categories, functor categories and block structure. In M. Nivat amd J. C. Reynolds, editor, *Algebraic Semantics*, pages 543–574. Cambridge University Press, 1985.

[ORW95] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified prescheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS93] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Intl. Symp, Gdansk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

[Ram96] John D. Ramsdell. personal communication, July 1996.

[RC$^+$86] Jonathan A. Rees, William C. Clinger, et al. Revised$^3$ report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[Sta94] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.

[TW96] Jerzy Tiuryn and Mitchell Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Trees in Algebra and Programming: CAAP'96, Proc. 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Heidelberg, and New York, April 1996. Springer-Verlag.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.