Axiomatizing Operational Equivalence in the presence of Side Effects

Ian A. Mason[*]
L.F.C.S., Edinburgh University

Carolyn Talcott[†]
Computer Science, Stanford University

1 Introduction

In this paper we present a formal system for deriving assertions about programs with side effects. The assertions we consider are of the following two forms: (i) e diverges (i.e. fails to reduce to a value), written $\uparrow e$; (ii) e_0 and e_1 are strongly isomorphic (i.e. reduce to the same value and have the same effect on memory up to production of garbage), written $e_0 \simeq e_1$. The e, e_j are expressions of a first-order Scheme- or Lisp-like language with the data operations *atom*, *eq*, *car*, *cdr*, *cons*, *setcar*, *setcdr*, the control primitives `let` and `if`, and recursive definition of function symbols.

The formal system we present defines a single-conclusion consequence relation $\Sigma \vdash \Phi$ where Σ is a finite set of constraints and Φ is an assertion. A constraint is an atomic or negated atomic formula in the first-order language consisting of equality, the unary function symbols *car* and *cdr*, the unary relation *atom*, and constants from the set of atoms, $\mathbb{A}$. Constraints have the natural first-order interpretation. The semantics of the formal system is given by a semantic consequence relation $\Sigma \models \Phi$ which is defined in terms of a class of memory models for assertions and constraints. The main results of this paper are

Theorem 1 *The deduction system is sound: if* $\Sigma \vdash \Phi$*, then* $\Sigma \models \Phi$*.*

Theorem 2 *The deduction system is complete for* Φ *not containing recursively defined function symbols: if* $\Sigma \models \Phi$*, then* $\Sigma \vdash \Phi$*.*

Operational equivalence [11, 15] abstracts the operational semantics of programs and is the basis for soundness results for program calculi and program transformation theories. Two expressions are operationally equivalent if they are indistinguishable in all program contexts. The importance of the strong isomorphism relation is that strong isomorphism relative to the empty set of constraints is the same as operational equivalence. Thus the formal system can be used for proving operational equivalence, and is complete for expressions which do not contain recursively defined function symbols.

From the rules of the formal system and the proof of completeness we obtain a decision procedure for the semantic consequence relation. This is an important step towards developing computer-aided deduction tools for reasoning about programs with memory.

Oppen [14] gives a decision procedure for the first-order theory of pure Lisp, i.e. the theory of *atom*, *car*, *cdr*, *cons* over acyclic list structures. Nelsen and Oppen [13] treat the quantifier-free case over possibly cyclic list structures. Neither treats updating operations. Boehm [1] defines a first-order theory for reasoning about programs in the language Russell which includes facilities for allocating and modifying memory. Some relative completeness results are given, but no decidable fragments are considered. The semantics of the full first-order Lisp-like language was studied in [5, 6]. Here the model-theoretic equivalence strong isomorphism was introduced and used as the basis for studying program equivalence. Many examples of proving program equivalence can be found in [8, 6, 7]. Felleisen [3] develops a calculus for reasoning about programs with memory, function abstractions and control abstractions. Mason and Talcott [9] give an alternative approach to treating programs with memory and function abstractions and develop the theory of operational equivalence for this case. More complete surveys of reasoning about programs with memory can be found in [5, 6, 7, 2, 3].

The remainder of this paper is organized as follows. We first define our language and its operational semantics. We then present the axioms and rules of

[*]Current address: Computer Science Department, Stanford University

[†]This research was partially supported by DARPA contract N00039-84-C-0211

the formal system. Following that we define memory models and semantic consequence and prove the soundness theorem. Finally we outline the proof of the completeness theorem. To do this we develop a syntactic representation of the operational semantics which is also useful for reasoning about programs in general. A full account of the proof may be found in [10].

We conclude this section with a summary of notational conventions. We use the usual notation for set membership and function application. Let Y, Y_0, Y_1 be sets. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $[y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. $[Y_0 \to Y_1]$ is the set of functions f with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f, $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$. $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbb{N}$.

2 The Semantics

In existing applicative languages there are two approaches to introducing objects with memory. We shall call these the Lisp approach and the ML approach. In the Lisp approach the semantics of lambda abstraction is modified so that upon application lambda variables are bound to newly allocated memory cells. Reference to a variable returns the contents of the cell and there is an assignment operation (`setq` or `set!`) for updating the contents of the cell bound to a variable. With this modified semantics one can no longer use beta-conversion to reason about program equivalence. For example in the program $((\lambda x \ldots \mathtt{setq}(x, n+1) \ldots)v)$ beta-conversion is not even meaningful, x cannot be substituted for by a value. Instead a cell must be allocated and x replaced by the cell name or labeled value. In the ML approach cells are added as a data type and operations are provided for creating cells and for accessing and modifying the contents. Reference to the contents of a cell must be made explicit. The semantics of lambda application is preserved and beta-value conversion remains a valid law for reasoning about programs. The Lisp approach provides a natural syntax since normally one wants to refer to the contents of a cell and not the cell itself. However the loss of the beta rule poses a serious problem for reasoning about programs. This approach also violates the principle of separating the mechanism for binding from that of allocation of memory [12]. Following the Scheme tradition, Felleisen [2] takes the Lisp approach to provide objects with memory. In order to obtain a reasonable calculus of programs, the programming language is extended to provide two sorts of lambda binding and an explicit dereferencing construct. There have been recent improvements in this calculus, but the problem of mixing binding and allocation is inherenent in the approach.

We take the ML approach to introducing objects with memory, adding primitive operations that create, access, and modify memory cells to the call-by-value lambda calculus. For brevity, we restrict our attention to expressions not containing recursively defined function symbols. The definitions and many of the intermediate results lift naturally to the full first-order language (see [6]).

We fix a countably infinite set of atoms, $\mathbb{A}$, with two distinct elements playing the role of booleans, `T` for *true* and `Nil` for *false*. We also fix a countable set $\mathbb{X}$ of variables disjoint from $\mathbb{A}$.

Definition 1 *The set of expressions, $\mathbb{E}$, is the smallest set containing $\mathbb{X} \cup \mathbb{A}$ and such that if $x \in \mathbb{X}$, $e_j \in \mathbb{E}$ for $j < 3$, $\delta_1 \in \mathbb{F}_1$, and $\delta_2 \in \mathbb{F}_2$ then $\mathtt{let}\{x := e_0\}e_1$, $\mathtt{if}(e_0, e_1, e_2)$, $\delta_1(e_1)$, and $\delta_2(e_1, e_2)$ are in $\mathbb{E}$. The unary memory operations are $\mathbb{F}_1 = \{atom, car, cdr\}$ and the binary memory operations are $\mathbb{F}_2 = \{eq, cons, setcar, setcdr\}$. We let $\mathbb{U}$ denote the set of value expressions $\mathbb{A} \cup \mathbb{X}$.* [1]

A more compact notation for the above standard inductive definition is given by the equation $\mathbb{E} = \mathbb{X} \cup \mathbb{A} \cup \mathtt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{E} \cup \mathtt{if}(\mathbb{E}, \mathbb{E}, \mathbb{E}) \cup \mathbb{F}_1(\mathbb{E}) \cup \mathbb{F}_2(\mathbb{E}, \mathbb{E})$, we shall freely use this notation in the sequel.

An expression describes a computation over S-expression memories — finite maps from (names of) cells to pairs of values, where a value is an atom or a cell. We call the value of a cell in a memory its contents. The memory operations are interpreted relative to a given memory as follows. *atom* is the characteristic function of the atoms, using the booleans `T` and `Nil`; *eq* tests whether two values are identical. *cons* takes two arguments, creates a new cell (extending the memory domain) whose contents is the pair of arguments, and returns the newly created cell. *car* and *cdr* return the first and second components of a cell. *setcar* and *setcdr* destructively alter an already existing cell. Given two arguments, the first of which

[1] We let $a, a_0, \ldots$ range over $\mathbb{A}$, $x, x_0, y, z \ldots$ range over $\mathbb{X}$, $e, e_0, \ldots$ range over $\mathbb{E}$ and $u, u_0, \ldots$ range over $\mathbb{U}$. The variable of a `let` is bound in the second expression, and the usual conventions concerning alpha conversion apply. We write $\mathrm{FV}(e)$ for the set of free variables of e. $\mathtt{seq}(e_0, \ldots, e_n)$ abbreviates $\mathtt{if}(e_0, \mathtt{seq}(e_1, \ldots, e_n), \mathtt{seq}(e_1, \ldots, e_n))$.

must be a cell, *setcar* updates the given memory so that the first component of the contents of its first argument becomes its second argument. *setcdr* similarly alters the second component. Thus memories can be constructed in which one or both components of a cell can refer to the cell itself.

To define the operational semantics we fix a countable set of (names of) cells, $\mathbb{C}$, disjoint from $\mathbb{A}$ and $\mathbb{X}$. $\mathbb{V} = \mathbb{A} \cup \mathbb{C}$ is the collection of storable memory values. The set of memories, $\mathbb{M}$, consists of finite maps from cells to pairs of values. Cells which appear in the range of a memory are assumed to lie in its domain. For each $n \in \mathbb{N}$ we also define a collection of n-ary memory objects, $\mathbb{O}^{(n)} \subseteq \mathbb{V}^n \times \mathbb{M}$, (elements of $\mathbb{O}^{(1)}$ are called objects, and we omit the superscript). The cells in the n-tuple component of a memory object must lie in the domain of its memory component. The set of environments or bindings, $\mathbb{B}$, is the collection of finite functions from $\mathbb{X}$ to $\mathbb{V}$. The set of descriptions of computations, $\mathbb{D}$, is a subset of $\mathbb{E} \times \mathbb{B} \times \mathbb{M}$. In a description the free variables of the expression must be in the domain of the environment, and cells in the range of the environment must be in the domain of the memory. [2]

The operational semantics of expressions is given by a reduction relation $\overset{*}{\mapsto}$ on descriptions. It is generated in the following manner. The action of the memory operations is given by the primitive reduction relation, $\rightharpoondown$, which is a subset of $(\mathbb{F}_1(\mathbb{O}) \times \mathbb{O}) \cup (\mathbb{F}_2(\mathbb{O}^{(2)}) \times \mathbb{O})$. $\overset{*}{\mapsto}$ is the reflexive transitive closure of the single-step relation $\mapsto$ which is defined in terms of reductions of *primitive expressions* and *reduction contexts*. The single-step reduction relation, $\mapsto$, is a subset of $(\mathbb{D} \times \mathbb{D})$, as is $\overset{*}{\mapsto}$. Finally, the evaluation relation, $\hookrightarrow$, is a subset of $(\mathbb{D} \times \mathbb{O})$. Evaluation is reduction composed with the operation converting value descriptions $(u; \beta; \mu)$ into memory objects.

The primitive reduction relation

$$\delta([v_0, \ldots, v_{n-1}]; \mu) \rightharpoondown v'; \mu'$$

is the least relation satisfying the following conditions.

$$atom(v; \mu) \rightharpoondown \begin{cases} \mathtt{T}; \mu & \text{if } v \in \mathbb{A} \\ \mathtt{Nil}; \mu & \text{otherwise} \end{cases}$$

$$car(c; \mu) \rightharpoondown v_0; \mu$$

$$cdr(c; \mu) \rightharpoondown v_1; \mu$$

$$eq([v_0, v_1]; \mu) \rightharpoondown \begin{cases} \mathtt{T}; \mu & \text{if } v_0 = v_1 \\ \mathtt{Nil}; \mu & \text{otherwise} \end{cases}$$

$$cons([v_0, v_1]; \mu) \rightharpoondown c; \mu\{c := [v_0, v_1]\}$$

$$setcar([c, v]; \mu) \rightharpoondown c; \mu\{c := [v, v_1]\}$$

$$setcdr([c, v]; \mu) \rightharpoondown c; \mu\{c := [v_0, v]\}$$

where in the cases for *car*, *cdr*, *setcar* and *setcdr* we assume that $c \in \mathrm{Dom}(\mu)$ and $\mu(c) = [v_0, v_1]$. The *cons* case holds for any c such that $c \notin \mathrm{Dom}(\mu)$. Although formally *cons* is multi-valued, the values differ only by renaming of cells and generally we will not distinguish them. Defining *cons* as a relation rather than a function which makes an arbitrary choice is the semantic analog of alpha conversion and greatly simplifies many definitions and proofs. If μ is a memory, then the function $car_\mu \in [\mathrm{Dom}(\mu) \to \mathbb{V}]$, is defined by $car_\mu(c) = v \leftrightarrow (\exists v')(\mu(c) = [v, v'])$, cdr_μ is defined analagously. Computation is a process of applying reductions to descriptions. The reduction to apply is determined by the unique decomposition of a non-value expression into a *reduction context* filled by a *primitive expression.*

Definition 2 *The set of primitive expressions, $\mathbb{E}_{\text{prim}}$, is defined to be* $\mathtt{if}(\mathbb{U}, \mathbb{E}, \mathbb{E}) \cup \mathtt{let}\{\mathbb{X} := \mathbb{U}\}\mathbb{E} \cup \mathbb{F}_1(\mathbb{U}) \cup \mathbb{F}_2(\mathbb{U}, \mathbb{U})$ *The set of contexts, ${}^{\varepsilon}\mathbb{E}$, is defined as usual, using the special symbol ε for holes. The set of reduction contexts, $\mathbb{R}$, is the subset of ${}^{\varepsilon}\mathbb{E}$ defined by* $\mathbb{R} = \{\varepsilon\} \cup \mathtt{let}\{\mathbb{X} := \mathbb{R}\}\mathbb{E} \cup \mathtt{if}(\mathbb{R}, \mathbb{E}, \mathbb{E}) \cup \mathbb{F}_1(\mathbb{R}) \cup \mathbb{F}_2(\mathbb{U}, \mathbb{R}) \cup \mathbb{F}_2(\mathbb{R}, \mathbb{E})$.[3]

Lemma 1 *If $e \in \mathbb{E}$ then either $e \in \mathbb{U}$ or e can be written uniquely as $R[\![e']\!]$ where R is a reduction context and $e' \in \mathbb{E}_{\text{prim}}$.*

The single-step reduction relation $\mapsto$ on $\mathbb{D}$ is defined in by the following three reductions, the first being (**beta**), the second (**if**), and the third (**delta**).

$$R[\![\mathtt{let}\{x := u\}e]\!]; \beta; \mu \mapsto R[\![e]\!]; \beta\{x := \beta(u)\}; \mu$$

$$R[\![\mathtt{if}(u, e_1, e_2)]\!]; \beta; \mu \mapsto \begin{cases} R[\![e_1]\!]; \beta; \mu & \text{if } \beta(u) \neq \mathtt{Nil} \\ R[\![e_2]\!]; \beta; \mu & \text{if } \beta(u) = \mathtt{Nil} \end{cases}$$

$$R[\![\delta(u_1, \ldots, u_n)]\!]; \beta; \mu \mapsto R[\![x]\!]; \beta\{x := v'\}; \mu'$$

where in the (**beta**) clause $x \notin \mathrm{Dom}(\beta)$ and in the (**delta**) clause $x \notin \mathrm{Dom}(\beta)$, $\delta \in \mathbb{F}_n$, $\delta([v_1, \ldots, v_n]; \mu) \rightharpoondown v'; \mu'$, and $v_i = \beta(u_i)$ for $1 \leq i \leq n$.

[2]We let $c, c_0, \ldots$ range over $\mathbb{C}$, $v, v_0, \ldots$ range over $\mathbb{V}$, $\mu, \mu_0, \ldots$ range over $\mathbb{M}$, $u; \mu, u_0; \mu_0, \ldots$ range over $\mathbb{O}$, $\beta, \beta_0, \ldots$ range over $\mathbb{B}$, and $e; \beta; \mu, e_0; \beta_0; \mu_0, \ldots$ range over $\mathbb{D}$. We use ";" in some notations, for example objects and descriptions, since some components of the these tuples are also collections (sets or tuples) and we wish to emphasize the outer level tuple structure. To simplify notation, we adopt the convention that $\beta(a) = a$ when $a \in \mathbb{A}$.

[3]We let E, E' range over ${}^{\varepsilon}\mathbb{E}$ and R range over $\mathbb{R}$. $E[\![e]\!]$ denotes the result of replacing any holes in E by e. Free variables of e may become bound in this process. We often adopt the usual convention that $[\![\ ]\!]$ denotes a hole. To avoid proliferation of brackets when dealing with composition of contexts we write $E; E'[\![e]\!]$ for $E[\![E'[\![e]\!]]\!]$ and similarly for longer composition chains.

A description $e;\beta;\mu \in \mathbb{D}$ evaluates to the object $v;\mu' \in \mathbb{O}$, if it reduces to a value description denoting that object: $e;\beta;\mu \hookrightarrow v;\mu'$ iff

$$(\exists u;\beta';\mu')(e;\beta;\mu \overset{*}{\mapsto} u;\beta';\mu' \wedge \beta'(u) = v)$$

A description is defined, written $\downarrow e;\beta;\mu$, just if

$$(\exists v;\mu' \in \mathbb{O})(e;\beta;\mu \hookrightarrow v;\mu').$$

A description is undefined, written $\uparrow e;\beta;\mu$, just if $\neg(\downarrow e;\beta;\mu)$. We identify a closed expression with the description consisting of it, the empty environment and the empty memory. Thus $\uparrow e$ abbreviates $\uparrow e;\emptyset;\emptyset$.

We define operational equivalence following [15].

Definition 3 *Two expressions are said to be operationally equivalent, written $e_0 \cong e_1$, if and only if for any closing context E, $E[\![e_0]\!]$ and $E[\![e_1]\!]$ are either both defined or both undefined.*

Alternatively one could define two closed expressions to be trivially equivalent if both are undefined, both return the same atom or both return cells. Then two expressions are operationally equivalent just if they are trivially equivalent in all closing contexts. This is the usual characterization of operational equivalence in the presence of basic data. Both definitions are equivalent in this setting since equality on basic data is computable. By definition operational equivalence is a congruence relation on expressions. However it is not necessarily the case that instantiations of equivalent expressions are equivalent even if the instantiation is defined. Explicitly: it is not the case that $\downarrow e$ and $e_0 \cong e_1$ implies $e_0\{x/e\} \cong e_1\{x/e\}$ for arbitrary variable x and expressions e, e_0, e_1.

3 The Formal System

In this section we present the language and rules of our formal system.

Definition 4 *The assertion language $\mathbb{L}$ and the constraint language $\mathcal{L}$ are as follows: $\mathbb{L} = (\mathbb{E} \simeq \mathbb{E}) \cup (\uparrow \mathbb{E})$ and $\mathcal{L} = (car(\mathbb{U}) = \mathbb{U}) \cup (cdr(\mathbb{U}) = \mathbb{U}) \cup (\mathbb{U} = \mathbb{U}) \cup \neg(\mathbb{U} = \mathbb{U}) \cup (atom(\mathbb{U})) \cup \neg(atom(\mathbb{U}))$.*

The set of constraints $\mathcal{L}$ is a subset of the atomic and negated atomic formulas in the first-order language consisting of equality, the unary function symbols *car* and *cdr*, the unary relation *atom*, and constants from $\mathbb{A}$. [4] We will freely use standard notions such as first-order satisfaction, $\models$. The theory $\mathrm{Th}(\mathbb{A})$ is defined by

$$\mathrm{Th}(\mathbb{A}) = \{atom(a), \neg(a = a') \mid a, a' \in \mathbb{A}, a \neq a'\}$$

To state the rules, as well as the side conditions on rules, we use the following notation. The result of pushing a context E through an assertion Φ is defined by

$$E[\![\Phi]\!] = \begin{cases} \uparrow E[\![e]\!] & \text{if } \Phi = \uparrow e \\ E[\![e_0]\!] \simeq E[\![e_1]\!] & \text{if } \Phi = e_0 \simeq e_1. \end{cases}$$

For $\vartheta \in \{car, cdr\}$, x is ϑ-less in Σ just if $\neg(\exists u \in \mathbb{U})(\Sigma \models \vartheta(x) = u)$ and $(\forall y \in \mathbb{X})((\vartheta(y) = u) \in \Sigma \to \Sigma \models \neg(x = y))$. $\mathrm{Dom}(\Sigma)$ is the subset of $\mathrm{FV}(\Sigma)$ defined by $\mathrm{Dom}(\Sigma) = \{x \in \mathrm{FV}(\Sigma) \mid \Sigma \models \neg atom(x)\}$. For each constraint $\varphi \in \mathcal{L}$ there is a corresponding assertion $T(\varphi) \in \mathbb{L}$ defined by

$$T(\varphi) = \begin{cases} eq(u_0, u_1) \simeq \mathtt{T} & \varphi \text{ is } u_0 = u_1 \\ eq(u_0, u_1) \simeq \mathtt{Nil} & \varphi \text{ is } \neg(u_0 = u_1) \\ atom(x) \simeq \mathtt{T} & \varphi \text{ is } atom(x) \\ atom(x) \simeq \mathtt{Nil} & \varphi \text{ is } \neg atom(x) \\ \vartheta(x) \simeq u & \varphi \text{ is } \vartheta(x) = u. \end{cases}$$

where $\vartheta \in \{car, cdr\}$.

Definition 5 *The consequence relation, $\vdash$, is the smallest relation that is closed under the rules given in figures 1 through 9.*

4 Soundness

In this section we present the semantics of our formal system. A model is an environment-memory pair such that cells in the range of the environment are in the domain of the memory. We let $\beta;\mu$, $\beta_0;\mu_0$, ... range over models. We begin by defining what it means for a model to satisfy an assertion or a constraint set. The semantic consequence relation between constraint sets and assertions is defined naturally in terms of these satisfaction relations.

Definition 6 *Two descriptions with the same model are strongly isomorphic, written $e_0;\beta;\mu \simeq e_1;\beta;\mu$, if exactly one of the following holds:*
1. *$\uparrow e_1;\beta;\mu$ and $\uparrow e_2;\beta;\mu$*
2. *There is a $v;\mu' \in \mathbb{O}$ with $\mathrm{Dom}(\mu) \subseteq \mathrm{Dom}(\mu')$ such that*

$$\bigwedge_{i<2} (\exists \mu_i \mid \mu' \subseteq \mu_i)(e_i;\beta;\mu \hookrightarrow v;\mu_i)$$

The model-theoretic equivalence strong isomorphism was introduced in [5] and used as the basis for studying program equivalence. The relation between

[4] We let $\varphi, \ldots$ range over $\mathcal{L}$, $\Phi, \ldots$ range over $\mathbb{L}$, and $\Sigma, \Sigma_0, \Delta, \ldots$ range over finite subsets of $\mathcal{L}$.

$$\text{(i)}\quad \Sigma \cup \{\varphi\} \vdash T(\varphi) \qquad \text{(ii)}\quad \frac{\Sigma \cup \{\varphi\} \vdash \Phi}{\Sigma \vdash \Phi} \quad \text{where in (ii) } \Sigma \cup \mathrm{Th}(\mathbb{A}) \models \varphi.$$

Figure 1: Structural rules

$$\text{(i)}\quad \frac{\Sigma \cup \{\varphi\} \vdash \Phi \qquad \Sigma \cup \{\neg\varphi\} \vdash \Phi}{\Sigma \vdash \Phi} \qquad \text{(ii)}\quad \frac{\Sigma \cup \{\vartheta(x) = z\} \vdash \Phi}{\Sigma \vdash \Phi}$$

where in (i) $\varphi \in \{atom(u), u_0 = u_1\}$ and in (ii) $\vartheta \in \{car, cdr\}, x \in \mathrm{Dom}(\Sigma)$, and $z \notin \mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma)$.

Figure 2: Left elimination

$$\text{(i)}\quad \Sigma \vdash e_0 \simeq e_0 \qquad \text{(ii)}\quad \frac{\Sigma \vdash e_0 \simeq e_1 \qquad \Sigma \vdash e_1 \simeq e_2}{\Sigma \vdash e_0 \simeq e_2} \qquad \text{(iii)}\quad \frac{\Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash e_1 \simeq e_0} \qquad \text{(iv)}\quad \frac{\Sigma \vdash eq(x,y) \simeq \mathtt{T}}{\Sigma \vdash x \simeq y}$$

Figure 3: Equivalence rules

$$\text{(i)}\quad \frac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash \uparrow e_1}{\Sigma \vdash e_0 \simeq e_1} \qquad \text{(ii)}\quad \frac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash \uparrow e_1} \qquad \text{(iii)}\quad \frac{\Sigma \vdash atom(x) \simeq \mathtt{T}}{\Sigma \vdash \uparrow \vartheta(x)}$$

$$\text{(iv)}\quad \frac{\Sigma \vdash atom(x) \simeq \mathtt{T}}{\Sigma \vdash \uparrow set\vartheta(x,y)} \quad \text{where in (iii),(iv) } \vartheta \in \{car, cdr\}.$$

Figure 4: Divergence rules

$$\text{(i)}\quad \frac{\Sigma \vdash \Phi}{\Sigma \vdash R[\![\Phi]\!]} \qquad \text{(ii)}\quad \Sigma \vdash R[\![\mathtt{if}(e_0, e_1, e_2)]\!] \simeq \mathtt{if}(e_0, R[\![e_1]\!], R[\![e_2]\!])$$

$$\text{(iii)}\quad \Sigma \vdash R[\![\mathtt{let}\{x := e_0\}e_1]\!] \simeq \mathtt{let}\{x := e_0\}R[\![e_1]\!] \quad \text{if } x \notin \mathrm{FV}(R).$$

Figure 5: Reduction context rules

$$\text{(i)}\quad \Sigma \vdash e \simeq \mathtt{let}\{x := e\}x \qquad \text{(ii)}\quad \Sigma \vdash e\{x/u\} \simeq \mathtt{let}\{x := u\}e \qquad \text{(iii)}\quad \Sigma \vdash \mathtt{if}(\mathtt{Nil}, e_0, e_1) \simeq e_1$$

$$\text{(iv)}\quad \Sigma \vdash \mathtt{seq}(e_0, e_1) \simeq \mathtt{let}\{x := e_0\}e_1 \quad x \notin \mathrm{FV}(e_1) \qquad \text{(v)}\quad \frac{\Sigma \vdash eq(u, \mathtt{Nil}) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{if}(u, e_0, e_1) \simeq e_0}$$

Figure 6: Rules concerning `let` and `if`

(i) $\Sigma \vdash \mathtt{let}\{x_0 := \mathit{cons}(\mathtt{T},\mathtt{T})\}\mathtt{let}\{x_1 := \mathit{cons}(\mathtt{T},\mathtt{T})\}e$

$\quad \simeq \mathtt{let}\{x_1 := \mathit{cons}(\mathtt{T},\mathtt{T})\}\mathtt{let}\{x_0 := \mathit{cons}(\mathtt{T},\mathtt{T})\}e$

(ii) $\Sigma \vdash \mathtt{seq}(e_0, \mathtt{let}\{x := \mathit{cons}(u_0, u_1)\}e_1) \simeq \mathtt{let}\{x := \mathit{cons}(u_0, u_1)\}\mathtt{seq}(e_0, e_1)$

(iii) $\dfrac{\Sigma \cup \Delta \vdash \Phi}{\Sigma \vdash \mathtt{let}\{x := \mathit{cons}(u_a, u_d)\}[\![\Phi]\!]}$

where in (ii) $x \notin \mathrm{FV}(e_0)$, and in (iii) $x \notin (\mathrm{FV}(\Sigma) \cup \{u_a, u_d\}) = Z$ and

$$\Delta = \{\neg \mathit{atom}(x), \mathit{car}(x) = u_a, \mathit{cdr}(x) = u_d, \neg(x = y) \mid y \in Z \cup (\mathrm{FV}(\Phi) - \{x\})\}$$

Figure 7: Rules for *cons*

(i) $\dfrac{\Sigma \vdash \mathit{eq}(x_0, x_2) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{seq}(\mathit{set}\vartheta(x_0, x_1), \mathit{set}\vartheta(x_2, x_3), e) \simeq \mathtt{seq}(\mathit{set}\vartheta(x_2, x_3), \mathit{set}\vartheta(x_0, x_1), e)}$

(ii) $\Sigma \vdash \mathtt{seq}(\mathit{set}\vartheta(x, y_0), \mathit{set}\vartheta(x, y_1)) \simeq \mathit{set}\vartheta(x, y_1)$ (iii) $\Sigma \vdash \mathtt{seq}(\mathit{set}\vartheta(x, y), x) \simeq \mathit{set}\vartheta(x, y)$

(iv) $\Sigma \vdash \mathtt{seq}(\mathit{setcdr}(x_0, x_1), \mathit{setcar}(x_2, x_3), e) \simeq \mathtt{seq}(\mathit{setcar}(x_2, x_3), \mathit{setcdr}(x_0, x_1), e)$

(v) $\Sigma \vdash \mathit{setcar}(\mathit{cons}(z, y), x) \simeq \mathit{cons}(x, y)$ (vi) $\Sigma \vdash \mathit{setcdr}(\mathit{cons}(x, z), y) \simeq \mathit{cons}(x, y)$

(vii) $\dfrac{\Sigma \cup \{\vartheta(x) = u_0\} \vdash \Phi}{\Sigma \cup \{\vartheta(x) = u_1\} \vdash \mathtt{seq}(\mathit{set}\vartheta(x, u_0), [\![\Phi]\!])}$

where $\vartheta \in \{\mathit{car}, \mathit{cdr}\}$ and in (vii) $x \in \mathrm{Dom}(\Sigma)$, and x is ϑ-less in Σ.

Figure 8: Rules for *setcar* and *setcdr*

If Γ is a context of the form

$\mathtt{let}\{z_1 := \mathit{cons}(\mathtt{T},\mathtt{T})\}\ldots\mathtt{let}\{z_n := \mathit{cons}(\mathtt{T},\mathtt{T}))\}$

$\quad \mathtt{seq}(\mathit{setcar}(z_1, u_1^a), \mathit{setcdr}(z_1, u_1^d), \ldots, \mathit{setcar}(z_n, u_n^a), \mathit{setcdr}(z_n, u_n^d), \varepsilon).$

and $\{z_1, \ldots, z_n\} \cap \mathrm{FV}(e) = \emptyset$, then $\Sigma \vdash \Gamma[\![e]\!] \simeq e$.

Figure 9: Garbage collection rule

strong isomorphism and operational equivalence is given by the following theorem.[5]

Theorem 3 *If $e_0, e_1 \in \mathbb{E}$, then $e_0 \cong e_1$ if and only if for every $\beta;\mu$ such that* $\mathrm{FV}(e_0, e_1) \subseteq \mathrm{Dom}(\beta)$ *we have $e_0;\beta;\mu \simeq e_1;\beta;\mu$.*

Definition 7 *The notion of a model satisfying an assertion, $\beta;\mu \models_{\mathbb{L}} \Phi$, is defined for* $\mathrm{FV}(\Phi) \subseteq \mathrm{Dom}(\beta)$ *by*

$$\beta;\mu \models_{\mathbb{L}} \Phi \;\leftrightarrow\; \begin{cases} \uparrow e;\beta;\mu & \text{if } \Phi = \uparrow e \\ e_0;\beta;\mu \simeq e_1;\beta;\mu & \text{if } \Phi = e_0 \simeq e_1. \end{cases}$$

The notion of a model satisfying a set of constraints $\beta;\mu \models_{\mathcal{L}} \Sigma$ is simply first-order satisfaction adapted to the memory structure framework. For any memory μ we define the corresponding first-order structure $\mathcal{M}_\mu$ by

$$\mathcal{M}_\mu = <\mathrm{Dom}(\mu) \cup \mathbb{A}, car_\mu, cdr_\mu, atom>$$

where $\mathrm{Dom}(\mu) \cup \mathbb{A}$ is the domain of $\mathcal{M}_\mu$, car_μ, cdr_μ are treated as binary relations, and *atom* is a unary relation. For $\beta \in \mathbb{B}$, $\varphi \in \mathcal{L}$ such that $\mathrm{FV}(\varphi) \subseteq \mathrm{Dom}(\beta)$ and $\mathrm{Rng}(\beta) \subseteq \mathrm{Dom}(\mu) \cup \mathbb{A}$ we write $\mathcal{M}_\mu \models \varphi\,[\beta]$ for the usual first-order satisfaction relation where $\varphi\,[\beta]$ is the interpretation of φ relative to the environment β, thought of as a Tarskian assignment. Thus

$$\mathcal{M}_\mu \models \varphi[\beta] \;\leftrightarrow\; \begin{cases} \beta(x) \in \mathbb{A} & \varphi \text{ is } atom(x) \\ \beta(x) \in \mathrm{Dom}(\mu) & \varphi \text{ is } \neg atom(x) \\ \beta(u_0) = \beta(u_1) & \varphi \text{ is } u_0 = u_1 \\ \beta(u_0) \neq \beta(u_1) & \varphi \text{ is } u_0 \neq u_1 \\ \vartheta_\mu(\beta(x)) = \beta(u) & \varphi \text{ is } \vartheta(x) = u \end{cases}$$

where $\vartheta \in \{car, cdr\}$. We say $\beta;\mu \models_{\mathcal{L}} \Sigma$ just if there is a $\beta' \supseteq \beta$ with $\mathrm{FV}(\Sigma) \subseteq \mathrm{Dom}(\beta')$ and $\mathrm{Rng}(\beta') \subseteq \mathbb{A} \cup \mathrm{Dom}(\mu)$ such that $\mathcal{M}_\mu \models \varphi\,[\beta']$ for $\varphi \in \Sigma$.

Definition 8 *The semantic consequence relation $\Sigma \models \Phi$ is defined by $\Sigma \models \Phi$ iff for every $\beta;\mu$ with* $\mathrm{FV}(\Phi) \subseteq \mathrm{Dom}(\beta)$ *we have*

$$\beta;\mu \models_{\mathcal{L}} \Sigma \;\rightarrow\; \beta;\mu \models_{\mathbb{L}} \Phi$$

A constraint set Σ is consistent just if $\beta;\mu \models_{\mathcal{L}} \Sigma$ for some model $\beta;\mu$.

The proof of the soundness theorem is now a routine matter of checking that the axioms are valid and that the rules preserve validity.

[5]This theorem holds for the full first-order language, not just the fragment with no recursively defined functions.

5 Completeness

In this section we outline the proof of the completeness theorem. We begin by developing a syntactic representation of descriptions and computation. We then present the key lemmas for the proof of completeness and the proof itself. Two forms of contexts feature in the syntactic analog of reduction: syntactic memory contexts and modifications. We define them in turn.

Definition 9 *The syntactic analog of a memory is a memory context, Γ, which is a context of the form*

$$\begin{array}{l} \mathtt{let}\{z_1 := cons(\mathtt{T},\mathtt{T})\} \ldots \mathtt{let}\{z_n := cons(\mathtt{T},\mathtt{T})\} \\ \quad \mathtt{seq}(setcar(z_1, u_1^a), setcdr(z_1, u_1^d), \\ \qquad \ldots, \\ \qquad setcar(z_n, u_n^a), setcdr(z_n, u_n^d), \\ \qquad \varepsilon). \end{array}$$

where $z_i \neq z_j$ when $i \neq j$.

In analogy to the semantic memories, we define the domain of Γ to be $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$. For Γ as above we define the functions $car_\Gamma, cdr_\Gamma \in [\mathrm{Dom}(\Gamma) \to \mathbb{U}]$ by $car_\Gamma(z_i) = u_i^a$ and $cdr_\Gamma(z_i) = u_i^d$. Two memory contexts are considered the same if they have the same domain and contents. Thus a memory context is determined by its domain and selector functions. We also define extension and updating operations on memory contexts. $\Gamma\{z := [u_{car}, u_{cdr}]\}$ is defined for $z \notin \mathrm{Dom}(\Gamma)$ to be the memory context Γ' obtained by extending Γ so that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$ and for $\vartheta \in \{car, cdr\}$, $\vartheta_{\Gamma'}(z) = u_\vartheta$. $\Gamma\{car(z) = u\}$ is defined for $z \in \mathrm{Dom}(\Gamma)$ to be the memory context Γ' obtained by altering Γ so that $car_{\Gamma'}(z) = u$. $\Gamma\{cdr(z) = u\}$ is defined similarly.

If $X \subseteq \mathbb{X} - \mathrm{Dom}(\Gamma)$, and $Z = \mathrm{FV}(\Sigma) \cup X \cup \mathrm{Dom}(\Gamma)$, then we define Σ_Γ^X — the set of constraints corresponding to the memory context Γ in the presence of free variables X — as follows,

$$\Sigma_\Gamma^X = \Sigma \cup \Delta$$

$$\Delta = \bigcup_{\substack{z \in \mathrm{Dom}(\Gamma) \\ u_\vartheta = \vartheta_\Gamma(z) \\ \vartheta \in \{car, cdr\} \\ y \in Z - \{z\}}} \{\neg atom(z), \vartheta(z) = u_\vartheta, \neg(z = y)\}$$

Definition 10 *A modification, M, is a context of the form*

$$\mathtt{seq}(set\vartheta_1(z_1, u_1), \ldots, set\vartheta_n(z_n, u_n), \varepsilon)$$

where $set\vartheta_i \in \{setcar, setcdr\}$ and $z_i = z_j$ implies $i = j$ or $set\vartheta_i \neq set\vartheta_j$.

We define $\mathrm{Dom}(M) = \{z_1, \ldots, z_n\}$ and $\vartheta_M(z_i) = u_i$ if $set\vartheta_i = set\vartheta$ for $\vartheta \in \{car, cdr\}$. Thus $\mathrm{Dom}(\vartheta_M) = \{z_i \in \mathrm{Dom}(M) \mid set\vartheta_i = set\vartheta\}$ for $\vartheta \in \{car, cdr\}$.

In analogy to the semantic reduction relations we define the relations $\rightharpoondown_\Sigma$, $\mapsto_\Sigma$, and $\overset{*}{\mapsto}_\Sigma$. In order to ensure that definitions are meaningful we introduce the notion of coherence. Roughly a constraint set and a memory-modification context are coherent (written $Coh(\Sigma, \Gamma; M)$) if $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(\Sigma) = \emptyset$, modifications in M are to elements of $\mathrm{Dom}(\Sigma)$, Σ decides equality on $\mathrm{Dom}(\Sigma)$, distinct elements of $\mathrm{Dom}(M)$ are provably distinct in Σ and Σ contains at most one *car* or *cdr* assertion for any z in $\mathrm{Dom}(\Sigma)$. (The last condition is a technicality to make various definitions and proofs simpler.) Note that coherence ensures that ϑ_M is single-valued modulo Σ equivalence. For Σ and $\Gamma; M$ such that $Coh(\Sigma, \Gamma; M)$ we define the relation

$$\Gamma; M[\![e]\!] \rightharpoondown_\Sigma \Gamma'; M'[\![e']\!]$$

analagously to the definition of $\rightharpoondown$. For example, letting $X = \mathrm{FV}(\Gamma; M[\![e]\!])$ and $\vartheta \in \{car, cdr\}$, the clauses for a representative selection of memory operations are as follows:
If $\Sigma_\Gamma^X \models \neg atom(u)$ then

$$\Gamma; M[\![atom(u)]\!] \rightharpoondown_\Sigma \Gamma; M[\![\mathtt{Nil}]\!].$$

If $(\exists u' \in \mathrm{Dom}(\vartheta_M))(\Sigma \models (u' = u))$ then

$$\Gamma; M[\![\vartheta(u)]\!] \rightharpoondown_\Sigma \Gamma; M[\![\vartheta_M(u)]\!].$$

If $\Sigma_\Gamma^X \cup \mathrm{Th}(\mathbb{A}) \models \neg(u_0 = u_1)$ then

$$\Gamma; M[\![eq(u_0, u_1)]\!] \rightharpoondown_\Sigma \Gamma; M[\![\mathtt{Nil}]\!].$$

If $z \in \mathbb{X} - (\mathrm{Dom}(\Gamma) \cup \mathrm{FV}(\Sigma) \cup X)$ then

$$\Gamma; M[\![cons(u_0, u_1)]\!] \rightharpoondown_\Sigma \Gamma\{z := [u_0, u_1]\}; M[\![z]\!].$$

For Σ and $\Gamma; M$ such that $Coh(\Sigma, \Gamma; M)$ we define the relation

$$\Gamma; M; R[\![e]\!] \mapsto_\Sigma \Gamma'; M'; R[\![e']\!],$$

analagously. For example

$$\Gamma; M; R[\![\mathtt{let}\{x := u\}e]\!] \mapsto_\Sigma \Gamma; M; R[\![e\{x/u\}]\!]$$

and assuming that $\Gamma; M[\![\delta(u_1, \ldots, u_n)]\!] \rightharpoondown_\Sigma \Gamma'; M'[\![u']\!]$, and $\mathrm{Dom}(\Gamma') - \mathrm{Dom}(\Gamma)$ is disjoint from $\mathrm{FV}(\Gamma; M; R[\![\delta(u_1, \ldots, u_n)]\!])$, and $\delta \in \mathbb{F}_n$. Then

$$\Gamma; M; R[\![\delta(u_1, \ldots, u_n)]\!] \mapsto_\Sigma \Gamma'; M'; R[\![u']\!].$$

For general use in reasoning about programs one would want to strengthen the definition of syntactic reduction by using full semantic satisfaction rather than first-order satisfaction in the side conditions. The weaker definition is adequate for proving completeness and simplifies the proof. Before we state the key lemmas, we require one last set of definitions. The rank of an expression $r(e)$ is just its size. The rank of an assertion $r(\Phi)$ is the maximum rank of the expressions occurring in Φ. $\mathrm{At}(\chi)$ is the set of atoms occuring in χ where χ is a finite set of expressions, constraints, and assertions. A *car-cdr* chain of length $\leq n$ is a reduction context of the form $\Theta = \vartheta_0(\vartheta_1(\ldots\vartheta_k(\varepsilon)\ldots))$ where $\vartheta_j \in \{car, cdr\}, j \leq k$, and $k < n$. For finite $X \subseteq \mathbb{X}$ and finite $A \subseteq \mathbb{A}$, we say Σ is n-complete w.r.t. $[X, A]$ just if for every Θ, Θ_0, *car-cdr* chains of length $\leq n$, and $y, y_0 \in X$, if $\Sigma \models \Theta[\![y]\!] = u$ and $\Sigma \models \Theta_0[\![y_0]\!] = u_0$, then

$$(\Sigma \models atom(u)) \vee (\Sigma \models \neg atom(u)),$$

for $\alpha \in A \cup \{\mathtt{T}, \mathtt{Nil}, u_0\}$

$$(\Sigma \models u = \alpha) \vee (\Sigma \models \neg(u = \alpha)),$$

and if $\Sigma \models \neg atom(u))$ then there are $u_a, u_d \in \mathbb{U}$ such that

$$(\Sigma \models car(u) = u_a) \wedge (\Sigma \models cdr(u) = u_d)$$

while if $\Sigma \models atom(u)$ then it is not the case that there are $u_a, u_d \in \mathbb{U}$ such that

$$(\Sigma \models car(u) = u_a) \vee (\Sigma \models cdr(u) = u_d)$$

The following five lemmas enable a straightforward proof of the completeness theorem.

Lemma 2 *If Σ is inconsistent, then $\Sigma \vdash \Phi$, for any $\Phi \in \mathbb{L}$.*

Lemma 3 *If $e \overset{*}{\mapsto}_\Sigma e'$, then $\Sigma \vdash e \simeq e'$.*

Lemma 4 *If $Coh(\Sigma, \emptyset)$ and Σ is $r(e)$-complete w.r.t. $[\mathrm{FV}(e), \mathrm{At}(\Sigma, e)]$, then there exists $\Gamma; M$ and an e' such that $e \overset{*}{\mapsto}_\Sigma \Gamma; M[\![e']\!]$ and exactly one of the following holds:*

1. *$e' = R[\![\vartheta(u)]\!], \vartheta \in \{car, cdr\}$ and $\Sigma \cup \mathrm{Th}(\mathbb{A}) \models atom(u)$.*

2. *$e' = R[\![set\vartheta(u_0, u_1)]\!], set\vartheta \in \{setcar, setcdr\}$ and $\Sigma \cup \mathrm{Th}(\mathbb{A}) \models atom(u_0)$.*

3. *$e' = u$, and $Coh(\Sigma, \Gamma; M)$.*

Lemma 5 *For any consistent Σ, finite $X \subseteq \mathbb{X}$, $\Phi \in \mathbb{L}$, and $n \in \mathbb{N}$ there exists $N \in \mathbb{N}$ and a family of constraint sets $\{\Sigma_i\}_{i<N}$ such that*

1. *Each Σ_i is n-complete w.r.t. $[X, \mathrm{At}(\Sigma_i, \Phi)]$, and $Coh(\Sigma_i, \emptyset)$.*

2. $(\forall\beta;\mu)(\beta;\mu \models_{\mathcal{L}} \Sigma \leftrightarrow (\exists i < N)(\beta;\mu \models_{\mathcal{L}} \Sigma_i))$

3. $\dfrac{\Sigma_i \vdash \Phi \quad i < N}{\Sigma \vdash \Phi}$ *is a derived rule.*

Lemma 6 *Let $e_i = \Gamma_i; M_i[\![u_i]\!]$ with $Coh(\Sigma, \Gamma_i; M_i)$ for $i < 2$. If $\Sigma \models e_0 \simeq e_1$ then $\Sigma \vdash e_0 \simeq e_1$.*

Proof (Completeness): Assume $\Sigma \models \Phi$. By lemma 2 we may assume that Σ is consistent. By lemma 5 it suffices to prove that $\Sigma \vdash \Phi$ under the added assumptions that $Coh(\Sigma, \emptyset)$ and Σ is $r(\Phi)$-complete w.r.t. $[\mathrm{FV}(\Phi), \mathrm{At}(\Sigma, \Phi)]$. By lemma 4 we have that for each e_i in Φ there exists $\Gamma_i; M_i$ and an e'_i such that $e_i \overset{*}{\mapsto}_\Sigma \Gamma_i; M_i[\![e'_i]\!]$ and exactly one of the following holds:

1. $e'_i = R_i[\![\vartheta_i(u_i)]\!]$, $\vartheta_i \in \{car, cdr\}$, and $\Sigma \cup \mathrm{Th}(\mathbb{A}) \models atom(u_i)$.

2. $e'_i = R_i[\![set\vartheta_i(u_i, u'_i)]\!]$, $set\vartheta_i \in \{setcar, setcdr\}$ and $\Sigma \cup \mathrm{Th}(\mathbb{A}) \models atom(u_i)$.

3. $e'_i = u_i$, and $Coh(\Sigma, \Gamma_i; M_i)$.

By lemma 3 we have $\Sigma \vdash e_i \simeq \Gamma_i; M_i[\![e'_i]\!]$ and by soundness we have $\Sigma \models e_i \simeq \Gamma_i; M_i[\![e'_i]\!]$. We consider two cases, depending on the nature of Φ.
Case $\Phi = \uparrow e$: Since Σ is consistent $e' \in \mathbb{U}$ is impossible.
In the other two cases we can show that $\Sigma \vdash \uparrow\Gamma; M[\![e']\!]$, and hence that $\Sigma \vdash \uparrow e$.
Case $\Phi = (e_0 \simeq e_1)$: We may assume that $\neg(\Sigma \models \uparrow e_i)$ since the case when $\Sigma \models \uparrow e_i$ follows directly from the previous case. Hence we have $\Sigma \vdash e_i \simeq \Gamma_i; M_i[\![u_i]\!]$ and $\Sigma \models e_i \simeq \Gamma_i; M_i[\![u_i]\!]$ for $i < 2$. Thus $\Sigma \models \Gamma_0; M_0[\![u_0]\!] \simeq \Gamma_1; M_1[\![u_1]\!]$ and by lemma 6 $\Sigma \vdash \Gamma_0; M_0[\![u_0]\!] \simeq \Gamma_1; M_1[\![u_1]\!]$.

□**Completeness**

6 Conclusion

We have presented a formal system for reasoning about equivalence of first-order Lisp- or Scheme-like programs that act on objects with memory. The semantics of the system is defined in terms of a notion of memory model derived from the natural operational semantics for the language. Equivalence is defined relative to classes of memory models defined by sets of constraints. The system is complete for programs that use only memory operations (no recursively defined functions, arithmetic operations, etc.). Thus the system can be seen to adequately express the semantics of memory operations. Presumably this could be extended to a relative completeness result for expressions built from memory and other algebraic operations, or for the full language, but we have not explored this possibility.

Equivalence in all models is the same as operational equivalence. Thus we have a means for reasoning about operational equivalence of programs. The formal system provides a richer language than operational equivalence since it provides a method for reasoning about conditional equivalence and equivalence with respect to restricted sets of contexts. This is essential for developing a theory of program transformations, since most of the interesting transformations are based on having additional information, i.e. on being able to restrict the contexts of use.

Implicit in the proof of completeness is a decision procedure for deciding when an expression is defined and whether two expressions are equivalent for all models of a set of constraints. There are three key algorithms. The first algorithm is an algorithm for deciding first-order consequence for constraints by a simple extension of an algorithm for putting a set of equations and inequations into a canonical form. The second algorithm generates a set of $r(e)$-complete constraints each of which completely determines the computational behavior of the expressions in question. The third algorithm finds a renaming of bound variables of a memory context that transforms one object expression into another that is equivalent modulo a set of constraints, or proves that no such bijection exists. Mindless application of these algorithms of course results in combinatorial explosion. An interesting open problem is to find strategies that are reasonably efficient for a useful class of queries and to incorporate this into a system for reasoning about programs.

Work is in progress to extend the formal system to a full higher-order Scheme-like language (with untyped lambda abstraction). Felleisen [2, 3] gives an equational calculus for reasoning about Scheme-like programs but such calculi do not deal adequately with conditional equivalence. The success of our approach in the first-order case depended on being able to define a semantics for conditional equivalence. In this case there is a natural model-theoretic equivalence (strong isomorphism) such that equivalence in all models is the same as operational equivalence. The existence of such a model-theoretic equivalence in the higher-order case remains an open question. The naive extension of the notion of strong isomorphism to the higher-order case does not work. Also operational equivalence in the first-order fragment does not imply equivalence in the higher-order language since non-atoms are no longer necessarily cells. Thus some refinement of the rules will be required.

Acknowledgements We would like to thank the following people for carefully reading earlier versions of this paper, and pointing out numerous mistakes and confusions: Louis Galbiati, Matthias Felleisen, Furio Honsell, and Elizabeth Wolf.

References

[1] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions, *ACM TOPLAS*, **7(4)**, pp. 637–655. 1985.

[2] Felleisen, M. The calculi of lambda-v-cs conversion: A syntactic theory of control and state in imperative higher-order programming languages, Ph.D. thesis, Indiana University. 1987

[3] Felleisen, M. λ-v-CS: An extended λ-calculus for Scheme, *Proceedings of the 1988 ACM conference on Lisp and functional programming,* pp. 72–85. 1988.

[4] Jørring, U. and Scherlis, W. L. Deriving and using destructive data types, *IFIP TC2 working conference on program specification and transformation*, (North–Holland). 1986.

[5] Mason, I. A. Equivalence of first order Lisp programs: proving properties of destructive programs via transformation, *Symposium on logic in computer science*, (IEEE), pp. 105–117. 1986.

[6] Mason, I. A. The semantics of destructive Lisp, CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University. 1986.

[7] Mason, I. A. Verification of programs which destructively alter data, *Science of Computer Programing,* **10**, pp. 177–210. 1988.

[8] Mason, I. A. and Talcott, C. L. Memories of S-expressions: Proving properties of Lisp-like programs that destructively alter memory, Department of Computer Science Report No. STAN-CS-85-1057, Stanford University, 1985.

[9] Mason, I. A. and Talcott, C. L. Programming, Transforming, and Proving with function abstractions and memories. Proceedings of the 16th EATCS Colloquium on Automata, Languages and Programming. Stresa. 1989.

[10] Mason, I. A. and Talcott, C. L. A Sound and Complete Axiomatization of Operational Equivalence between Programs with Memory. Department of Computer Science Report No. STAN-CS-89-????, Stanford University, 1989.

[11] Morris, J. H. Lambda calculus models of programming languages, Ph.D. thesis, Massachusetts Institute of Technology, 1968

[12] Mosses, P. A basic abstract semantic algebra, in: *Semantics of data types, international symposium, Sophia-Antipolis, June 1984, proceedings*, edited by G. Kahn, D. B. MacQueen, and G. Plotkin, Lecture notes in computer science, no. 173 (Springer, Berlin) pp. 87–108.

[13] Nelson, C. G. and Oppen, D. C. Fast decision procedures based on congruence closure, Computer Science Department Report STAN–CS–77–647, Stanford University, 1977.

[14] Oppen, D. C. Reasoning about recursively defined data structures, Computer Science Department Report STAN–CS–78–678, Stanford University, 1978.

[15] Plotkin, G. Call-by-name, call-by-value and the lambda calculus, *Theoretical Computer Science*, **1**, pp. 125–159, 1975.