Inferring the Equivalence of Functional Programs that Mutate Data

Ian Mason
Stanford University
IAM@CS.STANFORD.EDU

Carolyn Talcott
Stanford University
CLT@SAIL.STANFORD.EDU

1. Introduction

In this paper we study the constrained equivalence of programs with effects. In particular, we present a formal system for deriving such equivalences. Constrained equivalence is defined via a model theoretic characterization of operational, or observational, equivalence called strong isomorphism. Operational equivalence, as introduced by Morris [23] and Plotkin [27], treats programs as black boxes. Two expressions are operationally equivalent if they are indistinguishable in all program contexts. This equivalence is the basis for soundness results for program calculi and program transformation theories. Strong isomorphism, as introduced by Mason [14], also treats programs as black boxes. Two expressions are strongly isomorphic if in all memory states they return the same value, and have the same effect on memory (modulo the production of garbage). Strong isomorphism implies operational equivalence. The converse is true for first-order languages; it is false for full higher-order languages. However, even in the higher-order case, it remains an useful tool for establishing equivalence. Since strong isomorphism is defined by quantifying over memory states, rather than program contexts, it is a simple matter to restrict this equivalence to those memory states which satisfy a set of constraints. It is for this reason that strong isomorphism is a useful relation, even in the higher-order case.

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. The semantics of the formal system is given by a semantic consequence relation, $\Sigma \models \Phi$, defined in terms of a class of memory models for assertions and constraints.

The assertions we consider are of the following two forms: (i) e fails to return a value, written $\uparrow e$; (ii) e_0 and e_1 are strongly isomorphic, written $e_0 \simeq e_1$. In this paper we focus on the first-order fragment of a Scheme- or Lisp-like language, with data operations *cell*, *eq*, *car*, *cdr*, *cons*, *setcar*, *setcdr*, and control primitives `let` and `if`, and the recursive definition of function symbols.

A constraint set is a finite subset of atomic and negated atomic formulas in the first-order language consisting of equality, the unary function symbols *car* and *cdr*, the unary relation *cell*, and constants from the set of atoms, $\mathbb{A}$. Constraints have the natural first-order interpretation.

To illustrate the use of the formal system, we give three non-trivial examples of constrained equivalence assertions of well known list-processing programs. We also establish several metatheoretic properties of constrained equivalence and the formal system. The main results concerning the formal system are:

Theorem (Soundness): The deduction system is sound: $\Sigma \vdash \Phi \Rightarrow \Sigma \models \Phi$.

Theorem (Completeness): The deduction system is complete for Φ not containing recursively defined function symbols: $\Sigma \models \Phi \Rightarrow \Sigma \vdash \Phi$.

We also establish results relating the various notions of equivalence. Since both operational equivalence and strong isomorphism can be defined by quantifying over certain sets of contexts, it is of interest to compare these relations for various fragments of a full higher-order language. In this paper the two fragments of interest are the first-order fragment and the zero-order fragment (the subset of the first order fragment not containing recursively defined function symbols). In each of these fragments strong isomorphism ($\simeq_{\mathrm{zo}}$, $\simeq_{\mathrm{fo}}$) and operational equivalence ($\cong_{\mathrm{zo}}$, $\cong_{\mathrm{fo}}$) coincide. Furthermore, equivalence in one fragment coincides with equivalence in the other.

Theorem (Fragments):

$$\begin{tikzcd} e_0 \cong_{\mathrm{fo}} e_1 \arrow[r, Leftrightarrow] \arrow[d, Leftrightarrow] & e_0 \cong_{\mathrm{zo}} e_1 \arrow[d, Leftrightarrow] \\ e_0 \simeq_{\mathrm{fo}} e_1 \arrow[r, Leftrightarrow] & e_0 \simeq_{\mathrm{zo}} e_1 \end{tikzcd}$$

An early effort in the direction of equational theories for proving correctness of higher-order imperative programs is Demers and Donahue [6]. They present an equational proof system for deriving assertions about programs in the language Russell, an extension of the higher-order typed lambda calculus with cells and destructive cell operations. Their work is motivated by a desire to clarify the meaning of program constructs via an equational theory rather than an operational or denotational semantics. They consider one binary and three unary relations in their system. The unary relations express the legality, well-formedness and purity of expressions, while the binary relation represents a form of program equivalence. The simultaneous deduction of legality, well-formedness, purity and equivalence makes the rules very complex. No formal semantics for the proof system is given, and there are no formal results on the equational theory or its relationship to the original lambda calculus. Boehm [3] defines a first-order theory for reasoning about programs in the language Russell. Program constructs are defined by two classes of axioms. The first group concerns the nature of the value returned. The second group describes the effect on memory. Some relative completeness results are given, but no decidable fragments are considered. The underlying model theory is complex and rather cumbersome. The semantics of a full first-order Lisp-like language was studied in Mason [14, 13]. Here the model-theoretic equivalence strong isomorphism ($\simeq$) was introduced and used as the basis for studying program equivalence. Many examples of proving program equivalence can be found in Mason and Talcott [16, 14, 15, 18, 21, 20]. Felleisen [7] and Felleisen and Hieb [9] develop a calculus for reasoning about programs with memory, function abstractions and control abstractions. Mason and Talcott [17, 19] develop the theory of operational equivalence for programs with memory and function abstractions. More complete surveys of reasoning about programs with memory can be found in Mason [14, 13, 15] and Felleisen [7, 8]

The remainder of this paper is organized as follows. In §2. we define our first-order language, its operational semantics, the class of memory models, and the corresponding semantic consequence relations. In §3. we present the axioms and rules of the formal system and derive some simple consequences. In §4. we extend the formal system by adding an induction principle. This addition is necessary in order to prove properties of recursively defined functions. Three non-trivial examples of its use are provided. In §5. we prove the soundness theorem. §6. is devoted to 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. In §7. we relate the notions of operational equivalence and strong isomorphism in the first-order and zero-order fragments and their extension to include higher-order objects. In particular we present results that essentially characterize the difference between operational equivalence and strong isomorphism in the presence of higher-order objects. In §8. we present our conclusions and describe future directions of research.

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. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. $\mathbf{P}_\omega(Y)$ is the set of finite subsets of Y. $[Y_0 \to Y_1]$ is the set of total 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 set of natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbb{N}$.

2. The Operational Semantics

In existing applicative languages there are two mechanisms for, or approaches to, introducing objects with memory. We shall call these the *imperative* and *functional* approaches. In the imperative approach the semantics of lambda application is modified. Lambda variables are bound to unary memory cells. Variable cells are not first class citizens, and can not be explicitly manipulated. 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. In the functional approach cells are added as a data type, and operations are provided for creating cells, for accessing, and for modifying their contents. Reference to the contents of a cell must be made explicit. In the imperative approach one can no longer use beta-conversion to reason about program equivalence. Beta-conversion is not even meaningful in general, as variables that can be assigned cannot simply be replaced by values. For example the program $(\lambda x.\mathtt{seq}(\mathtt{setq}(x, 1), x))2$ evaluates to 1. The result of replacing all occurrences of x is an illegal program, while replacing only the final x alters the meaning of the program. Also, a variable x represents a value only if it is not assigned. Thus, whether or not $(\lambda x.e)x$ is equivalent to e depends on the context in which it occurs. To have a reasonable calculus one needs two sorts of variables: assignable and non-assignable. In the functional approach the semantics of lambda application is preserved, and beta-value conversion remains a valid law for reasoning about programs. The imperative 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 memory allocation (cf. Mosses [24]). Lisp and Scheme adopt both the imperative and the functional mechanisms for introducing memory. ML adopts only the functional mechanism. Following the Scheme tradition, Felleisen [7, 9] takes the imperative approach to introducing 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. In order prove several basic equivalences it is necessary to extend the calculus by meta principles (cf. the **safety rule** [7] thm 5.27, p.149). A key problem in developing such calculi is the trade-off between having a calculus rich enough to prove desired equivalences and having a calculus with nice theoretical properties such as Church-Rosser.

We take the functional approach to introducing objects with memory, adding primitive operations that create, access, and modify memory cells to the call-by-value lambda calculus. In the absence of higher-order objects, or structured data (tuples, records, ...) memories with cells that contain only a single value are not adequate for representing general list structures. In the higher-order case we could work with simple unary cell memories. Since we are working in the first-order case, we treat memories with binary cells. An alternative is to introduce structured data in the first-order case. We foresee no problem with doing this and plan to explore this approach in the future. Our work-to-date has focused attention on the memory aspects of computation.

2.1. The language

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 and for each $n \in \mathbb{N}$ a countable set, F_n, of n-ary function symbols. We assume the sets $\mathbb{A}$, $\mathbb{X}$, and F_n, for $n \in \mathbb{N}$, are pairwise disjoint. We let $\mathbf{F}_n$ denote the set of n-ary memory operations. The unary memory operations are $\mathbf{F}_1 = \{\mathit{cell}, \mathit{car}, \mathit{cdr}\}$, and the binary memory operations are $\mathbf{F}_2 = \{\mathit{eq}, \mathit{cons}, \mathit{setcar}, \mathit{setcdr}\}$. We let $\mathbb{F}_n$ abbreviate $\mathbf{F}_n \cup \mathrm{F}_n$, and $\mathbb{F}$ abbreviates $\bigcup_{n\in\mathbb{N}} \mathbb{F}_n$.

Definition ($\mathbb{U}$ $\mathbb{E}$): The set of value expressions, $\mathbb{U}$, is defined to be $\mathbb{X} \cup \mathbb{A}$. The set of expressions, $\mathbb{E}$, is defined to be the least set containing $\mathbb{U}$ such that if $x \in \mathbb{X}$, $n \in \mathbb{N}$, $e_j \in \mathbb{E}$ for $j \leq n$, and $\mathfrak{f} \in \mathbb{F}_n$, then $\mathtt{let}\{x := e_0\}e_1$, $\mathtt{if}(e_0, e_1, e_2)$, and $\mathfrak{f}(e_1, \ldots, e_n)$ are in $\mathbb{E}$.

This standard inductive definition is expressed more compactly by the following system of equations.

$$\mathbb{U} = \mathbb{X} \cup \mathbb{A}$$
$$\mathbb{E} = \mathbb{U} \cup \texttt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{E} \cup \texttt{if}(\mathbb{E}, \mathbb{E}, \mathbb{E}) \cup \bigcup_{n \in \mathbb{N}} \mathbb{F}_n(\mathbb{E}^n)$$

We will use this equational form of defining domains in the remainder of this paper. We also adopt the following notational conventions. $a, a_0, \ldots$ range over $\mathbb{A}$, $x, x_0, y, z, \ldots$ range over $\mathbb{X}$, $u, u_0, \ldots$ range over $\mathbb{U}$, and $e, e_0, \ldots$ range over $\mathbb{E}$. 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. We write $e\{x := e'\}$ to denote the expression obtained from e by replacing all free occurrences of x by e', avoiding the capture of free variables in e'. $\texttt{seq}(e)$ abbreviates e, while $\texttt{seq}(e_0, \ldots, e_n)$ abbreviates $\texttt{if}(e_0, \texttt{seq}(e_1, \ldots, e_n), \texttt{seq}(e_1, \ldots, e_n))$.

N.B. Throughout the remainder of this paper ϑ will range over $\{car, cdr\}$, and $set\vartheta$ will range over $\{setcar, setcdr\}$.

Definition (D): The set of (recursive function-) definition sets $\mathbf{D}$ is set of finite sequences of definitions of the form $<f(x_1, \ldots, x_n) \leftarrow e>$ where f is an n-ary function symbol.

$$\mathbf{D} = (\bigcup_{n \in \mathbb{N}} <\mathrm{F}_n(\mathbb{X}^n) \leftarrow \mathbb{E}>)^*$$

Let $\delta \in \mathbf{D}$ be a definition set. The defined functions of δ are those $f \in \mathrm{F}$ for which there is a definition $<f(x_1, \ldots, x_n) \leftarrow e>$ occurring in δ, for some $x_1, \ldots, x_n$, and e. The variables $(x_1, \ldots, x_n)$ are called the formal parameters of the definition, and e is called the body. A definition set, δ, is well-formed if no function symbol is defined more than once, for each $<f(x_1, \ldots, x_n) \leftarrow e>$ in δ, the variables $x_1, \ldots, x_n$ are distinct, $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$, and the function symbols occurring in e are among the defined functions of δ. We shall assume that definition sets are well-formed unless otherwise stated. Within a single definition the formal parameters are bound in the body, and we may freely rename them (subject to maintaining well-formedness).

2.2. Operational semantics

Expressions describe computations 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: *cell* is the characteristic function of the cells, 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 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 $\mathbb{A}$, $\mathbb{X}$, and $\mathbb{F}$. $\mathbb{V} = \mathbb{A} \cup \mathbb{C}$ 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 the collection of n-ary memory objects, $\mathbb{O}^{(n)} \subseteq \mathbb{V}^n \times \mathbb{M}$, (elements of $\mathbb{O}^{(1)}$ are called memory objects, or simply 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. Thus a memory object consists of a memory together with a sequence of values which *exist* in that memory. The interpretation of the memory operations will be given in terms of their action on memory objects. 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 lie in the domain of the environment, and cells in the range of the environment must lie in the domain of the memory. A description whose expression component is a value expression is called a value description. This is all summed up in the following definition.

Definition (Semantic Domains):

Values:	$\mathbb{V} = \mathbb{A} \cup \mathbb{C}$
Memories:	$\mathbb{M} = \{\mu \in [Z \to (Z \cup \mathbb{A})^2] \mid Z \in \mathbf{P}_\omega(\mathbb{C})\}$
Objects:	$\mathbb{O}^{(n)} = \{\bar{v};\mu \mid \mu \in \mathbb{M}, \bar{v} \in (\mathrm{Dom}(\mu) \cup \mathbb{A})^n\}$
Bindings:	$\mathbb{B} = \{\beta \in [X \to \mathbb{V}] \mid X \in \mathbf{P}_\omega(\mathbb{X})\}$
Descriptions:	$\mathbb{D} = \{e;\beta;\mu \mid \mathrm{FV}(e) \subseteq \mathrm{Dom}(\beta), \mathrm{Rng}(\beta) \subseteq \mathrm{Dom}(\mu) \cup \mathbb{A}, \mu \in \mathbb{M}\}$

We let $c, c_0, \ldots$ range over $\mathbb{C}$, $v, v_0, \ldots$ range over $\mathbb{V}$, $\mu, \mu_0, \ldots$ range over $\mathbb{M}$, $v;\mu, v_0;\mu_0, \ldots$ 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 ";" as an infix tupling construct in some notations, for example objects and descriptions, since some components of these tuples are also collections (sets or tuples), and we wish to emphasize the outer level tuple structure. We extend environments to value expressions by adopting the convention that $\beta(a) = a$ when $a \in \mathbb{A}$.

The operational semantics of expressions relative to a definition set, δ, is given by a reduction relation, $\overset{*}{\mapsto}$, on descriptions. It is generated in the following manner. $\overset{*}{\mapsto}$ is the reflexive transitive closure of the single-step relation, $\mapsto$, which is defined in terms of reductions of *redexes* and *reduction contexts*. The single-step reduction relation is a subset of $(\mathbb{D} \times \mathbb{D})$, as is $\overset{*}{\mapsto}$. The action of the memory operations is given by the primitive reduction relation, $\xrightarrow{\mathrm{p}}$, which is a subset of $\bigcup_{n \in \mathbb{N}}(\mathbf{F}_n(\mathbb{O}^n) \times \mathbb{O})$.

Finally, the evaluation relation, $\xrightarrow{}$, is a subset of $(\mathbb{D} \times \mathbb{O})$. The evaluation relation is the composition of the reduction relation with the operation of converting value descriptions $(u; \beta; \mu)$ into memory objects $(\beta(u); \mu)$. Officially, all of these relations should be parameterized by the definition set δ. Since we need only refer to a single arbitrary definition set, we will not make this parameter explicit.

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 *redex*.

Definition ($\mathbb{E}_{\mathrm{rdx}}$): The set of redexes, $\mathbb{E}_{\mathrm{rdx}}$, is defined as

$$\mathbb{E}_{\mathrm{rdx}} = \mathtt{if}(\mathbb{U}, \mathbb{E}, \mathbb{E}) \cup \mathtt{let}\{\mathbb{X} := \mathbb{U}\}\mathbb{E} \cup \bigcup_{n \in \mathbb{N}} \mathbb{F}_n(\mathbb{U}^n)$$

Definition (${}^{\varepsilon}\mathbb{E}$): The set of contexts, ${}^{\varepsilon}\mathbb{E}$, is defined inductively using the special symbol ε for holes:

$${}^{\varepsilon}\mathbb{E} = \{\varepsilon\} \cup \mathbb{X} \cup \mathbb{A} \cup \mathtt{let}\{\mathbb{X} := {}^{\varepsilon}\mathbb{E}\}{}^{\varepsilon}\mathbb{E} \cup \mathtt{if}({}^{\varepsilon}\mathbb{E}, {}^{\varepsilon}\mathbb{E}, {}^{\varepsilon}\mathbb{E}) \cup \bigcup_{n \in \mathbb{N}} \mathbb{F}_n({}^{\varepsilon}\mathbb{E}^n)$$

Definition ($\mathbb{R}$): The set of reduction contexts, $\mathbb{R}$, is the subset of ${}^{\varepsilon}\mathbb{E}$ defined by

$$\mathbb{R} = \{\varepsilon\} \cup \mathtt{let}\{\mathbb{X} := \mathbb{R}\}\mathbb{E} \cup \mathtt{if}(\mathbb{R}, \mathbb{E}, \mathbb{E}) \cup \bigcup_{n,m \in \mathbb{N}} \mathbb{F}_{n+m+1}(\mathbb{U}^n, \mathbb{R}, \mathbb{E}^m)$$

We let C, C' range over ${}^{\varepsilon}\mathbb{E}$ and R range over $\mathbb{R}$. $C[\![e]\!]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process. We often adopt the usual convention that $[\![\ ]\!]$ denotes a hole.

Lemma (Decomposition): 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 $e' \in \mathbb{E}_{\mathrm{rdx}}$.

Proof : By induction on the complexity of e. When $e \in \mathbb{U}$ the result is immediate. If $e = \mathtt{if}(e_0, e_1, e_2)$, then there are two possibilities. If $e_0 \in \mathbb{U}$, then let $R = \varepsilon$ and $e' = e$. Otherwise by the induction hypothesis $e_0 = R_0[\![e']\!]$ uniquely. Let $R = \mathtt{if}(R_0, e_1, e_2)$. The remaining cases are similar. □

Definition ($\mapsto$): The single-step reduction relation $\mapsto$ on $\mathbb{D}$ is defined by

(beta-value) $\quad R[\![\mathtt{let}\{x := u\}e]\!]; \beta; \mu \mapsto R[\![e]\!]; \beta\{x := \beta(u)\}; \mu \qquad x \notin \mathrm{Dom}(\beta)$

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

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

$\qquad$ if $x \notin \mathrm{Dom}(\beta)$, $\mathfrak{f} \in \mathbf{F}_n$, and $\mathfrak{f}([\beta(u_1), \ldots, \beta(u_n)]; \mu) \xrightarrow{\mathrm{p}} v'; \mu'$

(rec) $\quad R[\![f(u_1, \ldots, u_n)]\!]; \beta; \mu \mapsto R[\![e]\!]; \beta\{x_1 := \beta(u_1), \ldots, x_n := \beta(u_n)\}; \mu$

$\qquad$ if $x_1, \ldots, x_n \notin \mathrm{Dom}(\beta)$ and $<f(x_1, \ldots, x_n) \leftarrow e>$ in δ

The side condition in (**beta-value**) is to prevent free xs in $R[\![\ ]\!]$ from being trapped.

The primitive reduction relation, $\xrightarrow{\mathrm{p}}$, in (**delta**) is defined according to the nature of $\mathfrak{f} \in \mathbf{F}_n$.

Definition ($\xrightarrow{\mathrm{p}}$): The primitive reduction relation $\mathfrak{f}([v_0, \ldots, v_{n-1}]; \mu) \xrightarrow{\mathrm{p}} v'; \mu'$ is the least relation satisfying the following conditions.

$$cell(v; \mu) \xrightarrow{\mathrm{p}} \begin{cases} \mathtt{T}; \mu & \text{if } v \in \mathbb{C} \\ \mathtt{Nil}; \mu & \text{otherwise} \end{cases}$$

$$car(c; \mu) \xrightarrow{\mathrm{p}} v_0; \mu \qquad c \in \mathrm{Dom}(\mu) \text{ and } \mu(c) = [v_0, v_1]$$

$$cdr(c; \mu) \xrightarrow{\mathrm{p}} v_1; \mu \qquad c \in \mathrm{Dom}(\mu) \text{ and } \mu(c) = [v_0, v_1]$$

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

$$cons([v_0, v_1]; \mu) \xrightarrow{\mathrm{p}} c; \mu\{c := [v_0, v_1]\} \qquad \text{for any } c \text{ such that } c \notin \mathrm{Dom}(\mu)$$

$$setcar([c, v]; \mu) \xrightarrow{\mathrm{p}} c; \mu\{c := [v, v_1]\} \qquad c \in \mathrm{Dom}(\mu) \text{ and } \mu(c) = [v_0, v_1]$$

$$setcdr([c, v]; \mu) \xrightarrow{\mathrm{p}} c; \mu\{c := [v_0, v]\} \qquad c \in \mathrm{Dom}(\mu) \text{ and } \mu(c) = [v_0, v_1]$$

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.

Definition ($\rightharpoondown$): A description $e; \beta; \mu \in \mathbb{D}$ evaluates to the object $v; \mu' \in \mathbb{O}$, written $e; \beta; \mu \rightharpoondown v; \mu'$, if it reduces to a value description denoting that object.

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

As for primitive reductions, single-step reduction and evaluation are single-valued relations modulo renaming of cells.

Definition ($\downarrow \uparrow$): We write $\downarrow e; \beta; \mu$ just if $e; \beta; \mu$ returns a value (evaluates to some object), and $\uparrow e; \beta; \mu$ if $e; \beta; \mu$ fails to return a value.

$$\downarrow e; \beta; \mu \;\Leftrightarrow\; (\exists v; \mu' \in \mathbb{O})(e; \beta; \mu \rightharpoondown v; \mu')$$

$$\uparrow e; \beta; \mu \;\Leftrightarrow\; \neg(\downarrow e; \beta; \mu)$$

There are two ways for a description to fail to return a value: reduction terminating in an attempt to access or update the contents of a non-cell, and infinite reduction. The first case corresponds to an error, and is the only case which occurs in the absence of recursively defined functions. The treatment of errors and their relation to program equivalence is an important and non-trivial problem. We have chosen to consider errors as failure to produce a value in order to focus on properties of sharing and assignment.

2.3. Operational equivalence

We define operational equivalence following Plotkin [27]. For brevity, we identify a closed expression with the description consisting of it, the empty environment, and the empty memory. Thus $\uparrow e$ abbreviates $\uparrow e; \emptyset; \emptyset$.

Definition ($\cong$): Two expressions are said to be operationally equivalent, written $e_0 \cong e_1$, if and only if for any closing context C (i.e. one for which $\mathrm{FV}(C[\![e_0]\!]) = \emptyset = \mathrm{FV}(C[\![e_1]\!])$) $C[\![e_0]\!]$ and $C[\![e_1]\!]$ either both return a value or both fail to return a value.

$$(\forall C \in {}^{\varepsilon}\mathbb{E} \mid \mathrm{FV}(C[\![e_0]\!]) = \mathrm{FV}(C[\![e_1]\!]) = \emptyset)((\bigwedge_{i<2} \uparrow C[\![e_i]\!]) \vee (\bigwedge_{i<2} \downarrow C[\![e_i]\!]))$$

By definition operational equivalence is a congruence relation on expressions. Not all pairs of expressions are operationally equivalent. In particular, `T` and `Nil` are not operationally equivalent. It is important to note that being equivalent to a value is a much stronger condition than just returning a value. The usual characterization of operational equivalence in the presence of basic data is the following. Define two closed expressions to be trivially equivalent if both fail to return values, or 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. Both definitions are equivalent in this setting since equality on basic data is computable. These observations are summarized in the following lemma.

Lemma (opeq):

1. $e_0 \cong e_1 \;\Rightarrow\; (\forall C \in {}^{\varepsilon}\mathbb{E})(C[\![e_0]\!] \cong C[\![e_1]\!])$
2. $\neg(\mathtt{T} \cong \mathtt{Nil})$
3. $\downarrow e$ does not imply that $(\exists u)(u \cong e)$.
4. $e_0 \cong e_1 \;\Leftrightarrow\; (\forall C \in {}^{\varepsilon}\mathbb{E} \mid \mathrm{FV}(C[\![e_0]\!]) = \mathrm{FV}(C[\![e_1]\!]) = \emptyset)(C[\![e_0]\!] \cong^0 C[\![e_1]\!])$, where for closed expressions e_0', e_1' $e_0' \cong^0 e_1'$ iff

   $$(\bigwedge_{i<2} \uparrow e_i')$$

   or

   $$(\exists v_0; \mu_0, v_1; \mu_1)((\bigwedge_{i<2} e_i' \rightarrowtail v_i; \mu_i) \wedge ((v_0 = v_1 \wedge v_0, v_1 \in \mathbb{A}) \vee (v_0, v_1 \in \mathbb{C})))$$

Proof :

(1) Note that for any C, if C' is any closing context for $C[\![e_j]\!]$, $j \in 2$, then $C'[\![C]\!]$ is a closing context for e_j for $j \in 2$.

(2) The context $\mathtt{if}(\varepsilon, car(\mathtt{T}), \mathtt{T})$ will distinguish `T` and `Nil`.

(3) For example $cons(x, y)$, $\mathtt{if}(cell(x), setcar(x, y), x)$, and $\mathtt{if}(cell(x), car(x), x)$ always return values, but none are operationally equivalent to a value.

(4) The if direction is trivial. For the other direction suppose $(\bigwedge_{i<2} C[\![e_j]\!] \rightharpoondown v_j; \mu_j)$. If $v_0, v_1 \in \mathbb{A}$ and $v_0 \neq v_1$, then the context $\mathtt{if}(eq(v_0, C), car(\mathtt{T}), \mathtt{T})$ will distinguish the expressions. Similarly, if $v_0 \in \mathbb{A}$ and $v_1 \in \mathbb{C}$ then the context $\mathtt{if}(cell(C), \mathtt{T}, car(\mathtt{T}))$ will distinguish the expressions.

□

The impact of (**opeq.3**) is that in the case of programs with memory, returning a value is not an appropriate characterization of definedness. This is in contrast to the purely functional case and is due to the presence of *effects*.

It is not necessarily the case that instantiation of a variable by an value-returning-expression preserves equivalence. In other words 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. As a counter-example we have $eq(x, x) \cong \mathtt{T}$ but $eq(cons(\mathtt{T}, \mathtt{T}), cons(\mathtt{T}, \mathtt{T})) \cong \mathtt{Nil}$.

2.4. Strong Isomorphism and Constrained Equivalence

We now define strong isomorphism and constrained equivalence. The latter is achieved by defining what it means for a model, or memory state, 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 (model): 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.

Definition ($\simeq$): Two descriptions with the same model are strongly isomorphic, written $e_0; \beta; \mu \simeq e_1; \beta; \mu$, if both diverge or both evaluate to the same object up to production of cells not accessible from either the value or the domain of the original model:

1. $\uparrow e_1; \beta; \mu$ and $\uparrow e_2; \beta; \mu$, or
2. $(\exists v; \mu' \in \mathbb{O} \mid \mathrm{Dom}(\mu) \subseteq \mathrm{Dom}(\mu'))(\bigwedge_{i<2}(\exists \mu_i \mid \mu' \subseteq \mu_i)(e_i; \beta; \mu \rightharpoondown v; \mu_i))$

The relation between strong isomorphism and operational equivalence is given by the following theorem, proved in §7.

Theorem (Strong Isomorphism): 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 that $e_0; \beta; \mu \simeq e_1; \beta; \mu$.

By defining the assertion language $\mathbb{L} = (\mathbb{E} \simeq \mathbb{E}) \cup (\uparrow \mathbb{E})$, we can consider the strong isomorphism relation on descriptions as a notion of satisfaction, $\models_{\mathbb{L}}$.

Definition ($\mathbb{L}$):

$$\mathbb{L} = (\mathbb{E} \simeq \mathbb{E}) \cup (\uparrow \mathbb{E})$$

Definition ($\models_{\mathbb{L}}$): 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

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

We let $\Phi, \ldots$ range over $\mathbb{L}$.

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 *cell*, and constants from $\mathbb{A}$.

Definition ($\mathcal{L}$):

$$\mathcal{L} = (car(\mathbb{U}) = \mathbb{U}) \cup (cdr(\mathbb{U}) = \mathbb{U}) \cup (\mathbb{U} = \mathbb{U}) \cup \neg(\mathbb{U} = \mathbb{U}) \cup (cell(\mathbb{U})) \cup \neg(cell(\mathbb{U}))$$

We let $\varphi, \ldots$ range over $\mathcal{L}$, and $\Sigma, \Sigma_0, \Delta, \ldots$ range over $\mathbf{P}_\omega(\mathcal{L})$.

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 $\mathfrak{M}_\mu$ by

$$\mathfrak{M}_\mu = \langle \mathrm{Dom}(\mu) \cup \mathbb{A}, car_\mu, cdr_\mu, cell \rangle$$

where $\mathrm{Dom}(\mu) \cup \mathbb{A}$ is the domain of $\mathfrak{M}_\mu$, car_μ, cdr_μ are treated as binary relations (they correspond to the obvious partial functions with domain $\mathrm{Dom}(\mu)$), and *cell* is the unary relation corresponding to the set $\mathrm{Dom}(\mu)$.

Definition (ϑ_μ): If μ is a memory, then the binary relation, $\vartheta_\mu \subset \mathrm{Dom}(\mu) \times \mathbb{V}$, is defined by

$$car_\mu(c) = v \;\Leftrightarrow\; (\exists v')(\mu(c) = [v, v']) \quad \text{and} \quad cdr_\mu(c) = v \;\Leftrightarrow\; (\exists v')(\mu(c) = [v', v])$$

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 $\mathfrak{M}_\mu \models \varphi\,[\beta]$ 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

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

Definition ($\models_{\mathcal{L}}$): $\beta; \mu \models_{\mathcal{L}} \Sigma$ 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 $\mathfrak{M}_\mu \models \varphi\,[\beta']$ for every $\varphi \in \Sigma$.

Definition ($\Sigma \models \Phi$): The semantic consequence relation $\Sigma \models \Phi$ is defined by

$$\Sigma \models \Phi \;\Leftrightarrow\; (\forall \beta; \mu \mid \mathrm{FV}(\Phi) \subseteq \mathrm{Dom}(\beta))(\beta; \mu \models_{\mathcal{L}} \Sigma \;\Rightarrow\; \beta; \mu \models_{\mathbb{L}} \Phi).$$

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

In order to make the consequences of the above definition of satisfaction explicit, we state the following definitions and results, freely using standard notions such as first-order satisfaction, $\models$. Context will always disambiguate the overloading of the symbol $\models$. We do not distinguish between an element of $\mathbb{A}$ and the constant that denotes it. In particular we let $\mathfrak{A} = <\mathbb{A}, \mathit{cell}, a>_{a\in\mathbb{A}}$, and define the diagram of $\mathfrak{A}$ as in Chang and Keisler [5].

Definition (Diag($\mathfrak{A}$))**:** The diagram of the set of atoms, Diag($\mathfrak{A}$), is defined by

$$\mathrm{Diag}(\mathfrak{A}) = \{\neg \mathit{cell}(a), \neg(a = a') \mid a, a' \in \mathbb{A}, a \neq a'\}$$

Definition (Σ^m)**:** The memory structure theory, Σ^m, corresponding to a set of constraints, Σ, is defined by

$$\Sigma^m = \Sigma \cup \mathrm{Diag}(\mathfrak{A}) \cup \Sigma_{\mathrm{cell}}$$

$$\Sigma_{\mathrm{cell}} = \{\mathit{cell}(x) \mid (\exists u \in \mathbb{U})((\mathit{car}(x) = u) \in \Sigma \vee (\mathit{cdr}(x) = u) \in \Sigma)\}$$

Definition (T)**:** For each constraint, $\varphi \in \mathcal{L}$, there is a corresponding assertion, $T(\varphi) \in \mathbb{L}$, defined by

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

The following theorem is used in the completeness proof, in particular (**lemma 0**).

Theorem (Sat): For $\varphi \in \mathcal{L}$ we have $\Sigma^m \models \varphi \Leftrightarrow \Sigma \models T(\varphi)$.

Proof (Sat):

($\Leftarrow$) Assume that $\neg(\Sigma^m \models \varphi)$. Hence there exists a model

$$\mathfrak{B} = <B, \mathit{car}^{\mathfrak{B}}, \mathit{cdr}^{\mathfrak{B}}, \mathit{cell}^{\mathfrak{B}}>$$

and an assignment β such that $\mathfrak{B} \models (\Sigma^m \cup \{\neg\varphi\})[\beta]$. Now since $\mathrm{Diag}(\mathfrak{A}) \subseteq \Sigma^m$ we may assume that $\mathfrak{B} = <B_0 \cup \mathbb{A}, \mathit{car}^{\mathfrak{B}}, \mathit{cdr}^{\mathfrak{B}}, \mathit{cell}^{\mathfrak{B}}>$ and that β is the identity on $\mathbb{A}$. Furthermore, since $\mathbb{A}$ is infinite and Σ is finite, we may also assume that $\mathit{cell}^{\mathfrak{B}} = B_0$. Notice that we may alter the values of $\mathit{car}^{\mathfrak{B}}$ and $\mathit{cdr}^{\mathfrak{B}}$ on $B_0 - \{\beta(x) \mid x \in \mathrm{FV}(\Sigma \cup \{\varphi\})\}$ without effecting the satisfaction of Σ^m. Thus we may assume that $B_0 \subseteq \{\beta(x) \mid x \in \mathrm{FV}(\Sigma \cup \{\varphi\})\}$. Consequently define $\mu^{\mathfrak{B}}$ to be the memory with domain B_0 defined by $\mu^{\mathfrak{B}}(b) = [\mathit{car}^{\mathfrak{B}}(b), \mathit{cdr}^{\mathfrak{B}}(b)]$. Then it is easily checked that $\beta;\mu^{\mathfrak{B}} \models_{\mathcal{L}} \Sigma$ and $\neg(\beta;\mu^{\mathfrak{B}} \models_{\mathbb{L}} T(\varphi))$.

($\Rightarrow$) Suppose that $\neg(\Sigma \models T(\varphi))$. Then there exits a model $\beta;\mu$ with $\mathrm{Dom}(\beta) \supseteq \mathrm{FV}(\Sigma \cup \{\varphi\})$ such that $\beta;\mu \models_{\mathcal{L}} \Sigma$ and $\neg(\beta;\mu \models_{\mathbb{L}} T(\varphi))$. Now put $\mathfrak{B} = <\mathrm{Dom}(\mu) \cup \mathbb{A}, \mathit{car}^*_\mu, \mathit{cdr}^*_\mu, \mathrm{Dom}(\mu)>$ where $\vartheta^*_\mu(x) = \vartheta_\mu(x)$ when $x \in \mathrm{Dom}(\mu)$, and $\mathtt{Nil}$ otherwise. Then, since $\mathfrak{M}_\mu \models \Sigma[\beta]$ we have that $\mathfrak{B} \models \Sigma[\beta]$, and by construction $\mathfrak{B} \models \Sigma_{\mathrm{cell}}[\beta]$. Also since $\neg(\beta;\mu \models_{\mathbb{L}} T(\varphi))$, we know that $\mathfrak{B} \models \neg\varphi[\beta]$. Thus $\neg(\Sigma^m \models \varphi)$. $\square_{\mathbf{Sat}}$

3. The Formal System

In this section we present the rules of our formal system. To state the rules, as well as the side conditions on them, we make the following definitions.

Definition ($C[\![\Phi]\!]$): The result of pushing a context C through an assertion Φ is defined by

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

Definition (Cells(Σ)): Cells(Σ) is the subset of FV(Σ) defined by

$$\mathrm{Cells}(\Sigma) = \{x \in \mathrm{FV}(\Sigma) \mid \Sigma \models \mathit{cell}(x)\}.$$

If $x \in \mathrm{Cells}(\Sigma)$, then x must be interpreted as a cell.

Given a variable x and a constraint set Σ there are two distinct ways of adding information concerning the contents of (the cell associated with) x. The first is simply to add an assertion of the type $\vartheta(x) = u$. The second is to add an assertion of the form $x = y$, where y is a variable whose contents are already specified by Σ. To prevent this latter form of adding information, we introduce the notion of a variable x being ϑ-less in Σ. It is used to express the side conditions in the introduction and elimination rules for $\mathit{set}\vartheta$.

Definition (ϑ-less): x is ϑ-less in Σ just if $\neg(\exists u \in \mathbb{U})(\Sigma \models \vartheta(x) = u)$ and $(\forall y \in \mathbb{X})((\vartheta(y) = u) \in \Sigma \Rightarrow \Sigma \models \neg(x = y))$.

If x is ϑ-less in Σ, then the only way to consistently add information concerning $\vartheta(x)$ is by adding an assertion of the form $\vartheta(x) = u$. Furthermore, if x is ϑ-less in Σ, then we can add $\vartheta(x) = u$ to Σ without changing equality consequences of Σ. This is summed up in the following lemma.

Lemma (ϑ-lessness): Suppose $x \in \mathrm{Cells}(\Sigma)$ is ϑ-less in Σ. Then

0. Σ is consistent.
1. $(\forall u \in \mathbb{U})$ ($\Sigma \cup \{\vartheta(x) = u\}$ is consistent).
2. $(\Sigma \cup \{\vartheta(x) = u\} \models u_0 = u_1)$ iff $(\Sigma \models u_0 = u_1)$.

The following theorem justifies our liberal use of the first order satisfaction relation.

Theorem (Decidability): If $\Sigma \in \mathbf{P}_\omega(\mathcal{L})$ and $\varphi \in \mathcal{L}$, then it is decidable whether or not $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \varphi$.

Proof : This result follows from the fact that the quantifier-free theory of equality and uninterpreted function symbols is decidable, a result first proved by Ackermann [1]. Note that $(\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \varphi) \Leftrightarrow (\Sigma \cup D \models \varphi)$, where D is the restriction of Diag($\mathfrak{A}$) to the set of constants that appear in either Σ or φ. This has the consequence that $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \varphi$ can be decided in time quadratic in the size of Σ, (Nelson and Oppen [25]). □

3.1. The Rules

Definition ($\Sigma \vdash \Phi$): The consequence relation, $\vdash$, is the smallest relation on $\mathbf{P}_\omega(\mathcal{L}) \times \mathbb{L}$ that is closed under the rules given below.

Officially we should make δ a parameter of the consequence relation, but, as in the presentation of the operational semantics, we will not make this parameter explicit. The rules are partitioned into several groups. Each group of rules is given a label, for future reference, and members of the group are numbered. For example (**S.i**) refers to the first rule in the group of structural rules (the first group below). A rule has a (possibly empty) set of premisses and a conclusion. In the case that the set of premisses is non-empty the rule is displayed with a horizontal bar separating the premisses from the conclusion. A pair of rules that differ by interchanging premiss and conclusion is presented as a single rule with a double bar.

We begin with the structural rules. (**S.i**) asserts the obvious connection between a constraint, φ, and its translation into the assertion language, $T(\varphi)$. The second structural rule, (**S.ii**), allows us to replace any set of constraints with a logically equivalent set. It is used to put constraint sets into a form necessary for application of another rule — for example (**set.vii**). It also incorporates trivial facts concerning equality and the nature of the atoms.

Structural rules (S).

(i) $\Sigma \cup \{\varphi\} \vdash T(\varphi)$

(ii) $$\frac{\Sigma \cup \{\varphi\} \vdash \Phi}{\Sigma \vdash \Phi}\;(\text{double bar}) \qquad \text{assuming } \Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \varphi$$

The left elimination rules, (**L.i, L.ii**), enable one to reason by cases, while (**L.iii**) allows us to name the *car* and *cdr* of a cell.

Left elimination (L).

(i) $$\frac{\Sigma \cup \{cell(x)\} \vdash \Phi \qquad \Sigma \cup \{\neg cell(x)\} \vdash \Phi}{\Sigma \vdash \Phi}$$

(ii) $$\frac{\Sigma \cup \{\neg(u_0 = u_1)\} \vdash \Phi \qquad \Sigma \cup \{u_0 = u_1\} \vdash \Phi}{\Sigma \vdash \Phi}$$

(iii) $$\frac{\Sigma \cup \{\vartheta(x) = z\} \vdash \Phi}{\Sigma \vdash \Phi} \qquad \text{assuming that } x \in \mathrm{Cells}(\Sigma)\text{, and } z \notin \mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma)$$

The fact that $\simeq$ is an equivalence relation is expressed by the rules (**E.i, E.ii, E.iii**), while (**eq.i**) asserts that *eq* objects are also strongly isomorphic.

Equivalence rules (E).

(i) $\Sigma \vdash e_0 \simeq e_0$ (ii) $$\frac{\Sigma \vdash e_0 \simeq e_1 \quad \Sigma \vdash e_1 \simeq e_2}{\Sigma \vdash e_0 \simeq e_2}$$ (iii) $$\frac{\Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash e_1 \simeq e_0}$$

Rule concerning *eq* (eq).

(i) $$\frac{\Sigma \vdash eq(x,y) \simeq \mathtt{T}}{\Sigma \vdash x \simeq y}$$

The rules (**D**) express our treatment of error situations as a form of failure to return a value, i.e. as a form of divergence. Two divergent expressions are strongly isomorphic, (**D.i**). An expression strongly isomorphic to a divergent expression is itself divergent, (**D.ii**). The memory operations ϑ and $set\vartheta$ diverge when their first argument is not a cell, (**D.iii, D.iv**)

Divergence rules (D).

(i) $$\frac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash \uparrow e_1}{\Sigma \vdash e_0 \simeq e_1}$$ (ii) $$\frac{\Sigma \vdash \uparrow e_0 \qquad \Sigma \vdash e_0 \simeq e_1}{\Sigma \vdash \uparrow e_1}$$

(iii) $$\frac{\Sigma \vdash cell(x) \simeq \mathtt{Nil}}{\Sigma \vdash \uparrow \vartheta(x)}$$ (iv) $$\frac{\Sigma \vdash cell(x) \simeq \mathtt{Nil}}{\Sigma \vdash \uparrow set\vartheta(x,y)}$$

The reduction context rules express the extent to which constrained equivalence is a congruence. Note that all these rules are false for arbitrary contexts. In the case of $\uparrow$ assertions, (**R.i**) determines how this information is propagated.

Reduction context rules (R).

(i) $$\frac{\Sigma \vdash \Phi}{\Sigma \vdash R[\![\Phi]\!]}$$

(ii) $\Sigma \vdash R[\![\mathtt{if}(e_0,e_1,e_2)]\!] \simeq \mathtt{if}(e_0, R[\![e_1]\!], R[\![e_2]\!])$

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

The first `let` rule expresses the fact that the identity applied to an expression is equivalent to the expression itself. The second rule is simply beta-value reduction restricted to the first-order notion of a value.

Rules concerning `let` (let).

(i) $\Sigma \vdash e \simeq \mathtt{let}\{x := e\}x$ (ii) $\Sigma \vdash e\{x := u\} \simeq \mathtt{let}\{x := u\}e$

There are three rules concerning `if`. The first expresses a simple connection between `if` and `let`, recall that `seq`(e_0, e_1) abbreviates `if`(e_0, e_1, e_1). The second expresses the fact that an `if` reduces to its else clause when its test evaluates to `Nil`. The third expresses the fact that an `if` reduces to its then clause when the test evaluates to a non-`Nil` value.

Rules concerning `if` (if).

(i) $\Sigma \vdash \mathtt{seq}(e_0, e_1) \simeq \mathtt{let}\{x := e_0\}e_1$ assuming that $x \notin \mathrm{FV}(e_1)$

(ii) $\Sigma \vdash \mathtt{if}(\mathtt{Nil}, e_0, e_1) \simeq e_1$ (iii) $\dfrac{\Sigma \vdash eq(u, \mathtt{Nil}) \simeq \mathtt{Nil}}{\Sigma \vdash \mathtt{if}(u, e_0, e_1) \simeq e_0}$

There are three rules concerning *cons*. The first two assert that the time of allocation is irrelevant to the result of a *cons*. The third rule is essentially an introduction rule for *cons*. It expresses the fact that *cons* allocates a new cell with its arguments as the cells contents.

Rules for *cons* (cons).

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

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

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

subject to the condition that $x \notin \mathrm{FV}(e_0)$

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

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

There are seven rules concerning the $set\vartheta$ operations. (**set.i**) asserts that two successive $set\vartheta$ operations may be commuted, provided that they are altering distinct cells. (**set.ii**) asserts that successive modifications to the same cell is equivalent to simply carrying out the second modification. (**set.iii**) asserts that $set\vartheta$ returns as its value the modified cell. (**set.iv**) asserts that successive *setcar* and *setcdr* operations may always be commuted. (**set.v, set.vi**) assert that updating a newly allocated cell is equivalent to allocating it with the more recent contents. (**set.vii**) is an introduction and elimination rule for $set\vartheta$. The side condition that x is ϑ-less in Σ is needed to ensure, amongst other things, that the consistency of $\Sigma \cup \{\vartheta(x) = u_0\}$ is equivalent to the consistency of $\Sigma \cup \{\vartheta(x) = u_1\}$.

Rules for *setcar* and *setcdr* (set).

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

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

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

(iv) $\Sigma \vdash \mathtt{seq}(setcdr(x, y), setcar(w, z), e) \simeq \mathtt{seq}(setcar(w, z), setcdr(x, y), e)$

(v) $\Sigma \vdash setcar(cons(z, y), x) \simeq cons(x, y)$

(vi) $\Sigma \vdash setcdr(cons(x, z), y) \simeq cons(x, y)$

(vii) $\dfrac{\Sigma \cup \{\vartheta(x) = u_0\} \vdash \Phi}{\Sigma \cup \{\vartheta(x) = u_1\} \vdash \mathtt{seq}(set\vartheta(x, u_0), [\![\Phi]\!])}$ if $x \in \mathrm{Cells}(\Sigma)$ is ϑ-less

The garbage collection rule allows for the elimination of garbage — *cons* cells no longer accessible from the program text. Let Γ is a context of the form

$$\texttt{let}\{z_1 := cons(\texttt{T},\texttt{T})\}\ldots\texttt{let}\{z_n := cons(\texttt{T},\texttt{T}))\}$$
$$\texttt{seq}(setcar(z_1,u_1^a), setcdr(z_1,u_1^d), \ldots, setcar(z_n,u_n^a), setcdr(z_n,u_n^d), \varepsilon).$$

Garbage collection rule (G).

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

The unfolding rule (**U**) corresponds to the (**rec**) rule for single-step reduction. It is necessary in order to account for evaluation of recursively defined function symbols.

Unfolding rule (U).

(i) $\Sigma \vdash f(e_1,\ldots,e_n) \simeq \texttt{let}\{x_1 := e_1\}\ldots\texttt{let}\{x_n := e_n\}e$

where $<f(x_1,\ldots x_n) \leftarrow e>$ is in δ and x_i are chosen fresh.

3.2. Consequences

To illustrate the use of these rules we present some simple consequences.

Lemma (Mon): The provability relation is monotonic in the set of constraints, if $\Sigma \vdash \Phi$, then $\Sigma \cup \Sigma' \vdash \Phi$. In other words the rule (**monotonicity**) is derivable.

$$(\text{monotonicity})\quad \frac{\Sigma \vdash \Phi}{\Sigma \cup \Sigma' \vdash \Phi}$$

Proof (Mon): By induction on the length of proof and cases on the last rule applied. We consider the two most interesting cases.

(cons.iii) Assume that $x \notin \mathrm{FV}(\Sigma')$ and that we have derived

$$\Sigma \vdash \texttt{let}\{x := cons(u_a,u_d)\}[\![\Phi]\!]$$

where the last rule applied is (**cons.iii**). Then by induction hypothesis $\Sigma\cup\Sigma'\cup\Delta' \vdash \Phi$ where $\Delta' = \Delta \cup \{\neg(x = y) \mid y \in \mathrm{FV}(\Sigma')\}$. Hence by (**cons.iii**) we are done. $\square_{\mathbf{cons.iii}}$

(set.vii) Assume we have derived $\Sigma \cup \{\vartheta(x) = u_1\} \vdash \texttt{seq}(set\vartheta(x,u_0), [\![\Phi]\!])$ and the last rule applied is (**set.vii**) in the forward direction. Thus $x \in \mathrm{Cells}(\Sigma)$, x is ϑ-less in Σ, and $\Sigma \cup \{\vartheta(x) = u_0\} \vdash \Phi$. Using (**L.ii**) repeatedly, we may assume that for $\vartheta(z) = u \in \Sigma'$ we have either $\Sigma \cup \Sigma' \models z = x$ or $\Sigma \cup \Sigma' \models \neg(z = x)$. We first decompose Σ' into two sets Σ_x and Σ_0 so that x is ϑ-less in $\Sigma \cup \Sigma_0 \cup \Sigma_x$. Let

$$\Sigma_x = \{u = u_1 \mid (\exists z)(\vartheta(z) = u \in \Sigma' \wedge \Sigma \cup \Sigma' \models z = x)\}$$
$$\Sigma_0 = \Sigma' - \{\vartheta(z) = u \in \Sigma' \mid \Sigma \cup \Sigma' \models z = x\}.$$

Then

$$\Sigma \cup \Sigma_0 \cup \Sigma_x \cup \{\vartheta(x) = u_0\} \vdash \Phi \qquad \text{by induction hypothesis}$$
$$\Sigma \cup \Sigma_0 \cup \Sigma_x \cup \{\vartheta(x) = u_1\} \vdash \mathsf{seq}(set\vartheta(x, u_0), \Phi) \qquad \text{by } (\mathbf{set.vii})$$
$$\Sigma \cup \Sigma' \cup \{\vartheta(x) = u_1\} \vdash \mathsf{seq}(set\vartheta(x, u_0), \Phi) \qquad \text{by } (\mathbf{S}).$$

The proof for application of (**set.vii**) in the reverse direction is similar. $\square_{\mathbf{set.vii}}$

$\square_{\mathbf{Mon}}$

Lemma (Equality): Equals may be replaced by equals, if $\Sigma \models x = y$ and $\Sigma \vdash \Phi$, then $\Sigma \vdash \Phi\{x := y\}$. In other words the rule (**equality**) is derivable.

$$(\text{equality}) \quad \frac{\Sigma \cup \{x = y\} \vdash \Phi}{\Sigma \cup \{x = y\} \vdash \Phi\{x := y\}}$$

Proof (Equality): By induction on the length of proof. Again we consider only the interesting cases.

(S.i) If $\Sigma \models x = y$ then $\Sigma \cup \{\varphi\} \models \varphi\{x := y\}$.

(S.ii) If $\Sigma \models \varphi$ then $\Sigma \cup \{\varphi\} \models x = y \Leftrightarrow \Sigma \models x = y$.

(set.vii) Note that if $\Sigma \cup \{\vartheta(z) = u\} \models x = y$ and Σ is ϑ-less for z, then $\Sigma \models x = y$, and use (**S.ii**) in the case $z \in \{x, y\}$.

$\square_{\mathbf{Equality}}$

Lemma (Examples):

(i) $\{cell(x)\} \vdash setcar(x, car(x)) \simeq x$

(ii) $\Sigma \vdash cdr(cons(x, y)) \simeq y$

(iii) $\Sigma \vdash \mathsf{seq}(\mathsf{seq}(e_0, e_1), e_2) \simeq \mathsf{seq}(e_0, \mathsf{seq}(e_1, e_2))$

(iv) $\Sigma \vdash eq(x, y) \simeq eq(y, x)$

(v) $\Sigma \vdash eq(x, x) \simeq \mathtt{T}$

(vi) $\Sigma \vdash \neg cell(a) \simeq \mathtt{T}$

(vii) $\Sigma \vdash eq(a_0, a_1) \simeq \mathtt{Nil}$ assuming $a_0 \neq a_1$

(viii) $$\frac{\Sigma \vdash cell(x) \simeq \mathtt{Nil} \quad \Sigma \vdash cell(y) \simeq \mathtt{T}}{\Sigma \vdash eq(x, y) \simeq \mathtt{Nil}}$$

(ix) $\Sigma \vdash \mathtt{let}\{x_0 := \mathit{cons}(u_0^a, u_0^d)\}\mathtt{let}\{x_1 := \mathit{cons}(u_1^a, u_1^d)\}e$

$\simeq \mathtt{let}\{x_1 := \mathit{cons}(u_1^a, u_1^d)\}\mathtt{let}\{x_0 := \mathit{cons}(u_0^a, u_0^d)\}e$

provided $\{x_0, x_1\} \cap \mathrm{FV}(u_0^a, u_0^d, u_1^a, u_1^d) = \emptyset$

(x) $\Sigma \vdash \mathtt{let}\{y := e_0\}\mathtt{let}\{x := e_1\}e_2 \simeq \mathtt{let}\{x := \mathtt{let}\{y := e_0\}e_1\}e_2$ if $y \notin \mathrm{FV}(e_2)$

Proof (Examples):

(i) Let $\Sigma' = \{\mathit{cell}(x), \mathit{car}(x) = y\}$. Then by (**set.ii,iii**) we have

$$\Sigma' \vdash \mathtt{seq}(\mathit{setcar}(x,y), \mathit{setcar}(x,y)) \simeq \mathtt{seq}(\mathit{setcar}(x,y), x)$$

and by (**set.vii**) we have $\Sigma' \vdash \mathit{setcar}(x,y) \simeq x$. Now, using (**L,S,E,R**) we obtain $\Sigma \vdash \mathit{setcar}(x, \mathit{car}(x)) \simeq x$.

(ii) $\Sigma \vdash \mathtt{let}\{z := \mathit{cons}(x,y)\}\mathit{cdr}(z) \simeq \mathtt{let}\{z := \mathit{cons}(x,y)\}y$ by (**S.i, cons.iii**). Thus $\Sigma \vdash \mathit{cdr}(\mathit{cons}(x,y)) \simeq \mathtt{let}\{z := \mathit{cons}(x,y)\}y$ by (**R.iii,let.i**). The result now follows by (**E,G**) and the simple exercise showing that

$$\mathtt{let}\{z := \mathit{cons}(x,y)\}y \simeq \mathtt{let}\{z := \mathit{cons}(\mathtt{T},\mathtt{T})\}\mathtt{seq}(\mathit{setcar}(z,x), \mathit{setcdr}(z,y), y).$$

(iii) By (**R.ii**) and the definition of `seq`.

(iv) By (**L.ii,S,E**).

(v,vi,vii) By (**S**).

(viii) To show that $\Sigma \vdash \mathit{eq}(x,y) \simeq \mathtt{Nil}$ it suffices by (**L.ii, S.i**) to show that $\Sigma \cup \{x = y\} \vdash \mathit{eq}(x,y) \simeq \mathtt{Nil}$. By (**Equality, Mon, E, S.i**) and the assumptions we have that $\Sigma \cup \{x = y\} \vdash \mathtt{T} \simeq \mathtt{Nil}$. The result now follows by (**S.i, E**).

(ix) This is left to the reader. A similar derivation can be found in the proof of completeness.

(x) This is an instance of (**R.iii**).

□**Examples**

Expressions in contexts that correspond to the same memory construction are strongly isomorphic. A simple example of this is the lemma (**set absorption**)

Lemma (set absorption):

(1) $\vdash \mathtt{let}\{z := \mathit{cons}(x,y)\}\mathtt{seq}(\mathit{setcar}(z,w), e) \simeq \mathtt{let}\{z := \mathit{cons}(w,y)\}e$

(2) $\vdash \mathtt{let}\{z := \mathit{cons}(x,y)\}\mathtt{seq}(\mathit{setcdr}(z,w), e) \simeq \mathtt{let}\{z := \mathit{cons}(x,w)\}e$

Proof : We prove (**2**), the proof of (**1**) is identical.

$$\begin{aligned}
&\vdash \mathtt{let}\{z := \mathit{cons}(x,w)\}e \\
&\quad\simeq \mathtt{let}\{z := \mathtt{let}\{z' := \mathit{cons}(x,y)\}\mathit{setcdr}(z',w)\}e \qquad (\textbf{set.vi,R.iii,R.i}) \\
&\quad\simeq \mathtt{let}\{z' := \mathit{cons}(x,y)\}\mathtt{let}\{z := \mathit{setcdr}(z',w)\}e \qquad (\textbf{examples.x}) \\
&\quad\simeq \mathtt{let}\{z' := \mathit{cons}(x,y)\}\mathtt{let}\{z := \mathtt{seq}(\mathit{setcdr}(z',w),z')\}e \quad (\textbf{set.iii,cons.iii}) \\
&\quad\simeq \mathtt{let}\{z' := \mathit{cons}(x,y)\}\mathtt{seq}(\mathit{setcdr}(z',w),\mathtt{let}\{z := z'\}e) \quad (\textbf{R.iii,cons.iii}) \\
&\quad\simeq \mathtt{let}\{z := \mathit{cons}(x,y)\}\mathtt{seq}(\mathit{setcdr}(z,w),e) \qquad (\textbf{let.ii}) \text{ and renaming } z' \text{ to } z
\end{aligned}$$

□

3.3. Extensions

The inference system presented above is minimal by design in order to simplify the proof of completeness. The choice of rules is not necessarily the best if we want a basis that extends nicely. In particular a number of sound rules that are derivable in that system are no longer derivable when new rules are added. Two such rules are (**monotonicity**) and (**equality**). For the present we shall add these as official rules of inference:

Monotonicity (Mon).

$$\text{(i)} \quad \frac{\Sigma \vdash \Phi}{\Sigma \cup \Sigma' \vdash \Phi}$$

Equality (Eq).

$$\text{(i)} \quad \frac{\Sigma \cup \{x = y\} \vdash \Phi}{\Sigma \cup \{x = y\} \vdash \Phi\{x := y\}}$$

One difficulty with constrained equivalence is that it is not a congruence. One cannot, in general, place expressions equivalent under some non-empty set of constraints into an arbitrary program context and preserve equivalence. Informally, we say a context C does not invalidate a set of constraints Σ if the following principle is valid.

$$(\dagger) \quad \frac{\Sigma \vdash \Phi}{\Sigma \vdash C[\![\Phi]\!]}$$

There are some simple examples of this phenomena. The most trivial case is when Σ is empty. An easy case is when Σ contains only assertions of the form $\mathit{cell}(x)$, $\neg\mathit{cell}(x)$, $x = y$, or $\neg(x = y)$, and C is any context that does not trap the free variables of Σ. A somewhat harder case is when Σ any constraint set and C is of the form $\mathtt{let}\{x := e\}\varepsilon$ where (under constraint Σ) e has no write effect (evaluation of e will not execute any *setcar*s or *setcdr*s) and x is not free in Σ. Work of Lucassen and Gifford, [11, 12], makes progress in this direction, but needs to be modified if it is to apply in an untyped language. In what follows we shall adopt the most trivial version of context introduction as a rule.

Context Introduction (CI).

$$\text{(i)}\quad \frac{\emptyset \vdash \Phi}{\emptyset \vdash C[\![\Phi]\!]}$$

Another approach to overcoming this difficulty is to extend the system by adding a constraint propagation logic. Here one derives assertions of the form

$$\Sigma_0 \vdash C[\![\Sigma_1]\!].$$

The intended meaning of an assertion of the form $\Sigma_0 \vdash C[\![\Sigma_1]\!]$, is that if Σ_0 holds when evaluation of $C[\![\;]\!]$ begins, then Σ_1 will hold at the point in the program text where the hole appears. One consequence of the semantics of constraint propagation is that the following context propagation rule is valid.

$$\text{(CP)}\quad \frac{\Sigma_0 \vdash C[\![\Sigma_1]\!] \quad \Sigma_1 \vdash \Phi}{\Sigma_0 \vdash C[\![\Phi]\!]}$$

Notice that this rule is a variant of the invalid (†) principle. In particular the necessary side condition to validate (†) is that $\Sigma \vdash C[\![\Sigma]\!]$. This approach is taken in [21, 20].

4. Induction and Recursion

Although addition of the unfolding rule makes the inference system computationally adequate for the first-order case (§6. lemma 1.), it is inadequate to prove properties of recursively defined functions. As a first step to solving this problem we present a mechanism for introducing induction principles. In order to do this we define the notion of a ranked set (of memory objects).

Definition (ranked set): A ranked set is a pair (P, r) where $P \subseteq \mathbb{O}$ is a set of memory objects and r is a function from P to $\mathbb{N}$.

For example (*list*, *length*) is a ranked set. Here *list* is the set of memory objects $v;\mu$ such that $\{x := v\};\mu \models_{\mathbb{L}} cdr^n(x) \simeq \mathtt{Nil}$ for some $n \in \mathbb{N}$ (where $cdr^0(x)$ is x and $cdr^{n+1}(x)$ is $cdr(cdr^n(x))$), and *length* is the length function. Another example is (*sexp*, *size*) where *sexp* is the set of tree-like objects (no infinite *car*-*cdr* chains), and *size* is the number of reachable cells.

Let (P, r) be a ranked set of memory objects. To add P, r-induction to the inference system we first extend the language of constraints to include sets

$$P(\mathbb{U}) \cup (r(\mathbb{U}) = \mathbb{N})$$

Thus for each value expression $u \in \mathbb{U}$, $P(u)$ is a constraint, and for each natural number, n, $r(u) = n$ is also a constraint. The semantics of these constraints are given by the following.

- $\beta; \mu \models_{\mathcal{L}} P(u)$ just if $\beta(u); \mu \in P$.
- $\beta; \mu \models_{\mathcal{L}} r(u) = n$ just if $\beta(u); \mu \in P$ and $r(\beta(u); \mu) = n$.

The P, r-induction principle for constrained equivalence is the following. Let $\mathcal{E}$ be a family of equations of the form $e_0 \simeq e_1$ with distinguished free variable x. To show $\{P(x)\} \models \Phi$ for each Φ in $\mathcal{E}$ it suffices to show that $\{P(x), r(x) = n\} \models \Phi$ for each $n \in \mathbb{N}$, and each Φ in $\mathcal{E}$, assuming that $\{P(x), r(x) = m\} \models \Phi$ for $m < n$ and $\Phi \in \mathcal{E}$. This is made precise in the following theorem.

Theorem (P, r-induction):

$$(\forall n \in \mathbb{N})(\mathcal{I}_{P,r}(\mathcal{E}, n)) \Rightarrow (\forall \Phi \in \mathcal{E})(\{P(x)\} \models \Phi)$$

where $\mathcal{I}_{P,r}(\mathcal{E}, n)$ abbreviates

$$((\forall m < n)(\forall \Phi \in \mathcal{E})(\{P(x), r(x) = m\} \models \Phi)) \Rightarrow (\forall \Phi \in \mathcal{E})(\{P(x), r(x) = n\} \models \Phi)$$

Proof : Let P, r, $\mathcal{E}$ be as above and assume

† $(\forall n \in \mathbb{N})(\mathcal{I}_{P,r}(\mathcal{E}, n))$

We want to show that for any $\Phi \in \mathcal{E}$, and any model $\beta; \mu$ that $\beta; \mu \models_{\mathcal{L}} P(x)$ implies $\beta; \mu \models_{\mathbb{L}} \Phi$. Assume $\beta; \mu \models_{\mathcal{L}} P(x)$. Then by definition of rank function, $\beta; \mu \models_{\mathcal{L}} r(x) \simeq n$ for some $n \in \mathbb{N}$. We call this the rank of $\beta; \mu$. By induction on the rank of $\beta; \mu$ we have

$$(\forall m < n)(\forall \Phi \in \mathcal{E})(\{P(x), r(x) = m\} \models \Phi)$$

and by † and the definitions of $\models$ we are done. □

The reason for formulating the rule using a family equations is to overcome the lack of quantifiers. Thus we must strengthen the induction hypothesis by making explicit the necessary instances of the equations.

Given a particular ranked set, we can derive various sound induction rules using (P, r**-induction**). For lists we proceed as follows.

Definition ($\Delta^n_{list}(x)$ $\{x = [x_a, x_d]\}$): For $x \in \mathbb{X}$ and $n \in \mathbb{N}$ we define $\Delta^n_{list}(x)$ to be the set of constraints $\{list(x), length(x) = n\}$. Furthermore, if $x_a, x_d \in \mathbb{X}$, then we let $\{x = [x_a, x_d]\}$ denote the set $\{cell(x), car(x) = x_a, cdr(x) = x_d\}$.

We may write $\mathcal{E}(x)$ to emphasize the choice of distinguished variable and, assuming that y does not occur free in $\mathcal{E}(x)$, we write $\mathcal{E}(y)$ for the result of replacing x by y in each member of $\mathcal{E}(x)$. It is easy to see that the following is a sound rule.

List Induction Rule (LI).

$$\left\{\frac{\{\Delta^0_{list}(x), x = \mathtt{Nil} \vdash \Phi\}_{\Phi\in\mathcal{E}(x)} \qquad \coprod}{\{list(x)\} \vdash \Phi}\right\}_{\Phi\in\mathcal{E}(x)}$$

where

$$\coprod = \left\{\frac{\{\Delta^n_{list}(x_d) \vdash \Phi\}_{\Phi\in\mathcal{E}(x_d)}}{\Delta^{n+1}_{list}(x), \Delta^n_{list}(x_d), \{x = [x_a, x_d]\} \vdash \Phi}\right\}_{\Phi\in\mathcal{E}(x)}$$

and the notation $\{\Sigma \vdash \Phi\}_{\Phi\in\mathcal{E}(x)}$ denotes the set of judgements of the form $\Sigma \vdash \Phi$, for Φ a member of $\mathcal{E}(x)$. Within such a construct Φ is bound and can thus be renamed without changing the meaning. On the surface this is an infinitary rule. However, in practice the family of equations $\mathcal{E}$ are presented in a simple schematic form.

In the cases where $\mathcal{E}$ is finitely presented, or presented as a schemata, the induction rule (**LI**) can easily be encoded in, for example, the Edinburgh logical framework [10, 2], or reformulated in the style of Boyer and Moore [4].

We give three examples of the usage of the List induction principle. They serve to illustrate the variety of theorems provable. The proofs also provide examples of rather different families of equations. The third example best illustrates the need for non-trivial families of equations.

4.1. Iterative List Traversal

In this example we deal with two programs for appending lists. The first is the traditional pure program, *append*, that concatenates its first argument with its second, copying the top level list structure of its first argument in the process.

Definition (*append*)**:**

$$append(x, y) \leftarrow \mathtt{if}(eq(x, \mathtt{Nil}), y, cons(car(x), append(cdr(x), y)))$$

The problem with this definition of *append* is that to perform the *cons* in the non-trivial case we must first compute the result of *append*-ing the *cdr* of the first argument onto the second. This is easily seen to entail that *append* will use up stack proportional to the length of its first argument. The second program is an iterative version written using *setcdr*. It utilizes the destructive operations in the following way. Instead of waiting around for the result of doing the *append* of the *cdr* of the first argument before it can do the *cons*, it performs the *cons* with a, possibly, dummy *cdr* value and later on in the computation rectifies this haste. The result is a program that need not use any stack.

Definition (*iterative.append*)**:**

$$iterative.append(x, y) \leftarrow$$

$$\quad \texttt{if}(eq(x, \texttt{Nil}), y, \texttt{let}\{w := cons(car(x), y)\}\texttt{seq}(it.app(cdr(x), w, y), w))$$

$$it.app(x, w, y) \leftarrow$$

$$\quad \texttt{if}(eq(x, \texttt{Nil}),$$

$$\qquad x,$$

$$\qquad \texttt{let}\{z := cons(car(x), y)\}\texttt{seq}(setcdr(w, z), it.app(cdr(x), z, y)))$$

The following result could and should be taken as verification of the correctness of the *iterative.append* program, since we are reducing its behavior to that of a very simple program.

Theorem (append): $\{list(x)\} \vdash iterative.append(x, y) \simeq append(x, y)$

Before proving (**append**) we prove following lemma. It demonstrates that one can postpone setting the *cdr* of a newly created cell until the cell is referenced. This is the key property used in the optimization of *append* to *iterative.append*. An analogous result holds for the *car*. In stating the lemma we make use of our notation for pushing a context through an assertion defined in §3.

Lemma (delaying assignment): If $w \notin \mathrm{FV}(e)$ then

$$\vdash \texttt{let}\{w := cons(x, y)\}[\![\texttt{seq}(setcdr(w, z), e, e') \simeq \texttt{seq}(e, setcdr(w, z), e')]\!]$$

Proof (delaying assignment):

$$\vdash \texttt{let}\{w := cons(x, y)\}\texttt{seq}(setcdr(w, z), e, e') \simeq \texttt{let}\{w := cons(x, z)\}\texttt{seq}(e, e')$$

by (**set absorption**)

$$\simeq \texttt{seq}(e, \texttt{let}\{w := cons(x, z)\}e') \qquad \text{by } (\textbf{cons.ii})$$

$$\simeq \texttt{seq}(e, \texttt{let}\{w := cons(x, y)\}\texttt{seq}(setcdr(w, z), e'))$$

by (**set absorption, CI**)

$$\simeq \texttt{let}\{w := cons(x, y)\}\texttt{seq}(e, setcdr(w, z), e') \qquad \text{by } (\textbf{cons.ii})$$

□

Proof (append): We argue by cases, depending on whether $x = \texttt{Nil}$ or $\neg(x = \texttt{Nil})$.

$(x = \texttt{Nil})$ The result is trivially true when $x = \texttt{Nil}$ since:

$$\{x = \texttt{Nil}\} \vdash \mathit{iterative.append}(x, y)$$

$$\simeq \texttt{if}(\mathit{eq}(x, \texttt{Nil}), y, \texttt{let}\{w := \mathit{cons}(\mathit{car}(x), y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(x), w, y), w))$$

by (**U, let.ii**)

$$\simeq \texttt{if}(\texttt{T}, y, \texttt{let}\{w := \mathit{cons}(\mathit{car}(\texttt{Nil}), y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(\texttt{Nil}), w, y), w))$$

by (**S.i, R.i**)

$\simeq y$ by (**if.iii, S.i**)

$\simeq \mathit{append}(x, y)$ by identical reasoning

$\square_{\texttt{x=Nil}}$

$(\neg(x = \texttt{Nil}))$ In this case we use the following two lemmas (proved below)

Lemma (A):

$$\{x = [x_a, x_d]\} \vdash \mathit{cons}(x_a, \mathit{append}(x_d, y)) \simeq \mathit{append}(x, y)$$

Lemma (B):

$$\{\mathit{list}(x_d)\} \vdash \mathit{cons}(x_a, \mathit{append}(x_d, y)) \simeq \texttt{let}\{w := \mathit{cons}(x_a, y)\} \\ \texttt{seq}(\mathit{it.app}(x_d, w, y), w)$$

Then, letting $\Sigma_{list} = \mathit{list}(x), \mathit{list}(x_d), x = [x_a, x_d]$ we reason as follows. The first three steps unfold and simplify the definition of *iterative.append*. The next two steps evaluate $\mathit{car}(x)$ and $\mathit{cdr}(x)$ relative to Σ_{list}.

$$\Sigma_{list} \vdash \mathit{iterative.append}(x, y)$$

$$\simeq \texttt{if}(\mathit{eq}(x, \texttt{Nil}), y, \texttt{let}\{w := \mathit{cons}(\mathit{car}(x), y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(x), w, y), w))$$

using (**U, let.ii**) twice

$$\simeq \texttt{if}(\texttt{Nil}, y, \texttt{let}\{w := \mathit{cons}(\mathit{car}(x), y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(x), w, y), w))$$

by (**examples.viii, S.i, R.i**)

$\simeq \texttt{let}\{w := \mathit{cons}(\mathit{car}(x), y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(x), w, y), w))$ by (**if.ii**)

$\simeq \texttt{let}\{w := \mathit{cons}(x_a, y)\}\texttt{seq}(\mathit{it.app}(\mathit{cdr}(x), w, y), w))$ by (**S.i, R.i**)

$\simeq \texttt{let}\{w := \mathit{cons}(x_a, y)\}\texttt{seq}(\mathit{it.app}(x_d, w, y), w))$ by (**cons.iii, S.i, R.i**)

$\simeq \mathit{cons}(x_a, \mathit{append}(x_d, y))$ by (**B, monotonicity**)

$\simeq \mathit{append}(x, y)$ by (**A, monotonicity**)

$\square_{\neg(\texttt{x=Nil})}$

Proof (A): This is left as an exercise. $\square_{\mathbf{A}}$

Proof (B): Let

$$C_0[\![x_a]\!] = cons(x_a, append(x_d, y))$$
$$C_1[\![x_a]\!] = \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(it.app(x_d, w, y), w))$$

The proof is by List induction with induction variable x_d and $\mathcal{E}$ defined as follows:

$$\mathcal{E} = \{C_0[\![x_a]\!] \simeq C_1[\![x_a]\!] \mid x_a \in \mathbb{X} - \{w, y\}\}$$

(Base Case) Let $\Sigma_{base} = \Delta^0_{list}(x_d), x_d = \mathtt{Nil}$. Then

$$\Sigma_{base} \vdash \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(it.app(x_d, w, y), w))$$
$$\simeq \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(\mathtt{Nil}, w) \quad \text{by } (\mathbf{U, let.ii, if.ii, R.i, cons.iii})$$
$$\simeq \mathtt{let}\{w := cons(x_a, y)\}w \quad \text{by } (\mathbf{if.i, let.ii})$$
$$\simeq cons(x_a, y) \quad \text{by } (\mathbf{let.i})$$
$$\simeq cons(x_a, append(x_d, y)) \quad \text{as in the } x = \mathtt{Nil} \text{ proof, } (\mathbf{R.i})$$

$\square_{\mathbf{base}}$

(Induction Step) Let $\Sigma_{list} = \Delta^{n+1}_{list}(x_d), \Delta^n_{list}(x_{dd}), x_d = [x_{ad}, x_{dd}]$. Then

$$\Sigma_{list} \vdash it.app(x_d, w, y)$$
$$\simeq \mathtt{let}\{z := cons(car(x_d), y)\}\mathtt{seq}(setcdr(w, z), it.app(cdr(x_d), z, y))$$
by $(\mathbf{U, let.ii, S.i, if.iii, examples.viii})$
$$\simeq \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(setcdr(w, z), it.app(cdr(x_d), z, y), w)$$
by $(\mathbf{S.i, R.i})$

Now we introduce the context $\mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(\varepsilon, w)$ using (**cons.iii, R.i**) and the definition of seq, and permute the *cons*es using (**examples.ix**) to obtain

$$\Sigma_{list} \vdash \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(it.app(x_d, w, y), w)$$
$$\simeq \mathtt{let}\{z := cons(x_{ad}, y)\}$$
$$\mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(setcdr(w, z), it.app(cdr(x_d), z, y), w).$$

Using (**delaying assignment**) we have

$$\vdash \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(setcdr(w, z), it.app(cdr(x_d), z, y), w)$$
$$\simeq \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(it.app(cdr(x_d), z, y), setcdr(w, z), w).$$

Using (**cons.iii**) and (**examples.ix**) we introduce $\mathtt{let}\{z := cons(x_{ad}, y)\}\varepsilon$ and permute the *cons*es, and using (**S.i,cons.iii**) we evaluate $cdr(x_d)$ to x_{dd} obtaining

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{w := cons(x_a, y)\}\mathtt{seq}(it.app(x_d, w, y), w) \\
&\simeq \mathtt{let}\{w := cons(x_a, y)\} \\
&\qquad \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(it.app(x_{dd}, z, y), setcdr(w, z), w).
\end{aligned}$$

Rearranging and applying the induction hypothesis we have

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(it.app(x_{dd}, z, y), setcdr(w, z), w) \\
&\simeq \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(setcdr(w, \mathtt{seq}(it.app(x_{dd}, z, y), z)), w) \\
&\qquad \text{by } (\mathbf{R.ii}) \\
&\simeq \mathtt{seq}(setcdr(w, \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(it.app(x_{dd}, z, y), z)), w) \\
&\qquad \text{by } (\mathbf{R.iii}) \\
&\simeq \mathtt{seq}(setcdr(w, cons(x_{ad}, append(x_{dd}, y))), w) \\
&\qquad \text{by } (\mathbf{R.i}) \text{ and the induction hypothesis} \\
&\simeq \mathtt{let}\{w_d := append(x_d, y)\}setcdr(w, w_d) \\
&\qquad \text{by } (\mathbf{R.iii,\ A,\ R.i,\ set.iii,\ CI})
\end{aligned}$$

Finally, by (**cons.iii**), and permuting *cons*es we have

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{w := cons(x_a, y)\} \\
&\qquad \mathtt{let}\{z := cons(x_{ad}, y)\}\mathtt{seq}(setcdr(w, \mathtt{seq}(it.app(x_{dd}, z, y), z)), w) \\
&\simeq \mathtt{let}\{w_d := append(x_d, y)\}\mathtt{let}\{w := cons(x_a, y)\}setcdr(w, w_d) \\
&\simeq \mathtt{let}\{w_d := append(x_d, y)\}cons(x_a, w_d) \qquad \text{by } (\mathbf{set\ absorption,\ CI}) \\
&\simeq cons(x_a, append(x_d, y)) \qquad \text{by } (\mathbf{R.iii})
\end{aligned}$$

□**induction** □**B** □**append**

4.2. Copying and Modifying

In this example we treat the relationship between three programs, *copylist*, *reverse*, and *inplace.reverse*. *copylist* copies the top level or spine of its argument, which is assumed to be a list.

Definition (*copylist*)**:**

$$copylist(x) \leftarrow \mathtt{if}(eq(x, \mathtt{Nil}), x, cons(car(x), copylist(cdr(x))))$$

reverse produces a new list whose elements are the same as its arguments, except that they appear in the reverse order.

Definition (*reverse*)**:**

$$reverse(x) \leftarrow rev(x, \texttt{Nil})$$

$$rev(x, y) \leftarrow \texttt{if}(eq(x, \texttt{Nil}), y, rev(cdr(x), cons(car(x), y)))$$

inplace.reverse also produces list whose elements are the same as its arguments, except that they appear in the reverse order. However it constructs this list by re–using the cells in top level or spine of its argument. It is called *nreverse* in most dialects of Lisp.

Definition (*inplace.reverse*)**:**

$$inplace.reverse(x) \leftarrow in.rev(x, \texttt{Nil})$$

$$in.rev(x, y) \leftarrow \texttt{if}(eq(x, \texttt{Nil}), y, in.rev(cdr(x), setcdr(x, y)))$$

A property of *inplace.reverse* is that when applied to the result of copying the top level structure of a list it is equivalent to *reverse*. This is typical of the theorems that can be proved about destructive versions of list and other structure manipulating functions. It states that if we can prove that the top level structure of the argument list is accessible only to the reverse program, then we are free to optimize by doing an inplace reverse.

Theorem (inplace reverse):

$$\{list(x)\} \vdash inplace.reverse(copylist(x)) \simeq reverse(x)$$

Proof (inplace reverse): By (**U**) and the rules concerning `let` we have

$$\{list(x)\} \vdash inplace.reverse(copylist(x)) \simeq \texttt{let}\{z := copylist\} in.rev(z, \texttt{Nil})$$

$$\{list(x)\} \vdash reverse(x) \simeq rev(x, \texttt{Nil})$$

Thus we need only show

$$\{list(x)\} \vdash \texttt{let}\{z := copylist(x)\} in.rev(z, y) \simeq rev(x, y).$$

This is done by List-induction with $\mathcal{E}$ is the set of equations consisting of the equation to be proved, together with all variants obtained by replacing y by any variable other than x.

$$\mathcal{E} = \{\texttt{let}\{z := copylist(x)\} in.rev(z, y) \simeq rev(x, y) \;\big|\; y \in \mathbb{X} - \{x\}\}$$

(Base Case) The base case is trivial. Letting $\Sigma_{base} = \Delta^0_{list}(x), x = \mathtt{Nil}$, we have

$$\begin{aligned}
\Sigma_{base} \vdash\ & \mathtt{let}\{z := \mathit{copylist}(x)\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z := \mathtt{if}(\mathit{eq}(x, \mathtt{Nil}), x, \mathit{cons}(\mathit{car}(x), \mathit{copylist}(\mathit{cdr}(x))))\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z := \mathtt{Nil},\}\mathit{in.rev}(z, y) \\
&\simeq y \\
&\simeq \mathit{rev}(x, y)
\end{aligned}$$

$\square_{\mathbf{base}}$

(Induction Step) Let $\Sigma_{list} = \Delta^{n+1}_{list}(x), \Delta^n_{list}(x_d), x = [x_a, x_d]$. Firstly observe that by the definition of *copylist*, the assumptions regarding x and the `let` rules we have

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{z := \mathit{copylist}(x)\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z := \mathit{cons}(x_a, \mathit{copylist}(x_d))\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z_d := \mathit{copylist}(x_d)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{in.rev}(z, y)
\end{aligned}$$

Now unfolding and simplifying the definition of *in.rev*, evaluating $\mathit{cdr}(z)$ and using the laws for *setcdr* and `seq` we have

$$\begin{aligned}
\{z = [x_a, z_d]\} \vdash\ & \mathit{in.rev}(z, y) \\
&\simeq \mathit{in.rev}(\mathit{cdr}(z), \mathit{setcdr}(z, y)) \\
&\simeq \mathtt{seq}(\mathit{setcdr}(z, y), \mathit{in.rev}(z_d, z))
\end{aligned}$$

Using the above, (**cons.iii**), and (**set absorption**)

$$\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{in.rev}(z, y) \simeq \mathtt{let}\{z := \mathit{cons}(x_a, y)\}\mathit{in.rev}(z_d, z)$$

Using context introduction and *cons* rules we can show that:

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{z_d := \mathit{copylist}(x_d)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z := \mathit{cons}(x_a, y)\}\mathtt{let}\{z_d := \mathit{copylist}(x_d)\}\mathit{in.rev}(z_d, z)
\end{aligned}$$

Finally by the above, the induction hypothesis, (**cons.iii**) and the definition of *rev* we have

$$\begin{aligned}
\Sigma_{list} \vdash\ & \mathtt{let}\{z := \mathit{copylist}(x)\}\mathit{in.rev}(z, y) \\
&\simeq \mathtt{let}\{z := \mathit{cons}(x_a, y)\}\mathtt{let}\{z_d := \mathit{copylist}(x_d)\}\mathit{in.rev}(z_d, z) \\
&\simeq \mathtt{let}\{z := \mathit{cons}(x_a, y)\}\mathit{rev}(x_d, z) \\
&\simeq \mathit{rev}(x, y)
\end{aligned}$$

$\square_{\mathbf{induction}}$ $\square_{\mathbf{ir}}$

4.3. Copying and Delaying

This example is an instance of a class of useful theorems about delaying structure traversal (cf. [18]). The idea is that by copying a structure, applications of functions that traverse the structure but have no effect on interior components can be postponed until the results are needed — i.e. moved across arbitrary computations that intervene between the call and the first use of the result. We prove the delay theorem for the case of *append*.

Theorem (delaying append): If z not free in e_0 and w is fresh, then

$$\begin{aligned}&\{\mathit{list}(x)\} \vdash \mathtt{let}\{z := \mathit{append}(x, y)\}\mathtt{let}\{x_0 := e_0\}e_1\\ &\quad\simeq \mathtt{let}\{w := \mathit{copylist}(x)\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1\end{aligned}$$

Note that even though the statement of the theorem does not explicitly involve effects, the evaluation of the arbitrary expression e_0 can have quite dramatic effects.

Proof (delaying append): Let

$$\begin{aligned}C_0[\![e_0, e_1]\!] &= \mathtt{let}\{z := \mathit{append}(x, y)\}\mathtt{let}\{x_0 := e_0\}e_1\\ C_1[\![e_0, e_1]\!] &= \mathtt{let}\{w := \mathit{copylist}(x)\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1\end{aligned}$$

The proof is by List-induction taking $\mathcal{E}$ to be:

$$\mathcal{E} = \{C_0[\![e_0, e_1]\!] \simeq C_1[\![e_0, e_1]\!] \mid e_0, e_1 \in \mathbb{E}, z \notin \mathrm{FV}(e_0)\}$$

(Base Case) Let $\Sigma_{base} = \Delta^0_{list}(x), x = \mathtt{Nil}$. Then

$$\Sigma_{base} \vdash \mathtt{let}\{z := \mathit{append}(x, y)\}\mathtt{let}\{x_0 := e_0\}e_1 \simeq \mathtt{let}\{z := y\}\mathtt{let}\{x_0 := e_0\}e_1$$

by unfolding and simplifying the call to *append*

$\simeq \mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := y\}e_1$ by (**let.ii, CI**)

$\simeq \mathtt{let}\{w := x\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1$

by (**U, S.i, R.i, let.ii, CI**)

$\simeq \mathtt{let}\{w := \mathit{copylist}(x)\}\mathtt{let}\{y := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1$

by the definition of *copylist* and assumptions regarding x

$\square_{\mathbf{base}}$

(Induction Step) Again let $\Sigma_{list} = \Delta^{n+1}_{list}(x), \Delta^{n}_{list}(x_d), x = [x_a, x_d]$.

$\Sigma_{list} \vdash \mathtt{let}\{z := \mathit{append}(x, y)\}\mathtt{let}\{x_0 := e_0\}e_1$

$\simeq \mathtt{let}\{z_d := \mathit{append}(x_d, y)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathtt{let}\{x_0 := e_0\}e_1$

by unfolding and simplifying the call to *append*

$\simeq \mathtt{let}\{z_d := \mathit{append}(x_d, y)\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}e_1$

by (**cons.ii, CI**)

$\simeq \mathtt{let}\{w_d := \mathit{copylist}(x_d, y)\}$

$\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z_d := \mathit{append}(w_d, y)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}e_1$

by the induction hypothesis

$\simeq \mathtt{let}\{w_d := \mathit{copylist}(x_d, y)\}$

$\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{cons}(x_a, \mathit{append}(w_d, y))\}e_1$

by (**R.i, let.i, CI**)

$\simeq \mathtt{let}\{w_d := \mathit{copylist}(x_d, y)\}$

$\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(\mathit{cons}(x_a, w_d), y))\}e_1$

by the definition of *append* and (**CI**)

$\simeq \mathtt{let}\{w_d := \mathit{copylist}(x_d, y)\}$

$\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{w := \mathit{cons}(x_a, w_d)\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1$

by (**let.i, R.i, CI**)

$\simeq \mathtt{let}\{w_d := \mathit{copylist}(x_d, y)\}$

$\mathtt{let}\{w := \mathit{cons}(x_a, w_d)\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1$

by (**cons.ii, CI**)

$\simeq \mathtt{let}\{w := \mathit{copylist}(x, y)\}\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{z := \mathit{append}(w, y)\}e_1$

by definition of *copylist*

$\square$**induction** $\square$**da**

5. Soundness

In this section we state and prove the soundness theorem.

Theorem (Soundness): If $\Sigma \vdash \Phi$ then $\Sigma \models \Phi$.

Proof (Soundness): It suffices to show that each rule preserves soundness, i.e. soundness of the premisses implies soundness of the conclusion. We restrict our attention to those rules for which this result is non-trivial. The proofs for the remaining rules are either trivial or else minor variations on the ones given.

Lemma (S): $\Sigma \cup \{\varphi\} \models T(\varphi)$ for $\varphi \in \mathcal{L}$.

Proof (S): Suppose $\beta; \mu \models_{\mathcal{L}} \Sigma \cup \{\varphi\}$ and without loss of generality that $\mathrm{FV}(\Sigma \cup \{\varphi\}) \subseteq \mathrm{Dom}(\beta)$. Then by definition $\beta; \mu \models_{\mathcal{L}} \varphi$. This together with the definition of T is sufficient to force that $\beta; \mu \models_{\mathbb{L}} T(\varphi)$. $\Box_{\mathbf{S}}$

Lemma (L): Suppose that $\vartheta \in \{car, cdr\}$, $x \in \mathrm{Cells}(\Sigma)$, and $z \notin \mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma)$. Then

$$\frac{\Sigma \cup \{\vartheta(x) = z\} \models \Phi}{\Sigma \models \Phi}$$

Proof (L): Suppose that $\neg(\Sigma \models \Phi)$. Then without loss of generality we may assume that there is a $\beta; \mu$ such that $\mathrm{Dom}(\beta) = \mathrm{FV}(\Sigma) \cup \mathrm{FV}(\Phi)$ with $\beta; \mu \models_{\mathcal{L}} \Sigma$ and $\neg(\beta; \mu \models_{\mathbb{L}} \Phi)$. Since $z \notin \mathrm{FV}(\Sigma) \cup \mathrm{FV}(\Phi)$ we have that $\beta\{z := \vartheta_\mu(\beta(x))\}; \mu \models_{\mathcal{L}} \Sigma \cup \{\vartheta(x) = z\}$ and $\neg(\beta\{z := \vartheta_\mu(\beta(x))\}; \mu \models_{\mathbb{L}} \Phi)$. Thus $\neg(\Sigma \cup \{\vartheta(x) = z\} \models \Phi)$. $\Box_{\mathbf{L}}$

Lemma (cons): Suppose that $\Phi \in \mathbb{L}, x \notin (\mathrm{FV}(\Sigma) \cup \{u_a, u_d\}) = Z$ and

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

Then

$$\frac{\Sigma \cup \Delta \models \Phi}{\Sigma \models \mathtt{let}\{x := cons(u_a, u_d)\}[\![\Phi]\!]}$$

Proof (cons): Suppose that $\mathrm{FV}(\Sigma) \subseteq \beta$, $x \notin \mathrm{Dom}(\beta)$ and that $\beta; \mu \models_{\mathcal{L}} \Sigma$. Furthermore assume that $\neg(\beta; \mu \models \mathtt{let}\{x := cons(u_a, u_d)\}[\![\Phi]\!])$. Thus choosing $c \notin \mathrm{Dom}(\mu)$ and letting $\beta'; \mu' = \beta\{x := c\}; \mu\{c := [\beta(u_a), \beta(u_d)]\}$ we have that $\neg(\beta'; \mu' \models \Phi)$. Consequently it suffices to show that $\beta'; \mu' \models_{\mathcal{L}} \Sigma \cup \Delta$. This is routine. $\Box_{\mathbf{cons}}$

Lemma (set): Suppose that $\Phi \in \mathbb{L}, x \in \mathrm{Cells}(\Sigma)$ and x is cdr-less in Σ. Then

$$\frac{\Sigma \cup \{cdr(x) = u_0\} \models \Phi}{\Sigma \cup \{cdr(x) = u_1\} \models \mathtt{seq}(setcdr(x, u_0), [\![\Phi]\!])}$$

Proof (set): Pick $\beta; \mu$ such that $\beta; \mu \models_{\mathcal{L}} \Sigma$, $\mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma) \cup \{x, u_i\} \subseteq \mathrm{Dom}(\beta)$, $\beta(x) = c$ and for $i < 2$ put

$$\begin{aligned} \Sigma_i &= \Sigma \cup \{cdr(x) = u_i\} \\ \mu_i &= \mu\{c := [car_\mu(c), \beta(u_i)]\} \\ \Phi_0 &= \Phi \\ \Phi_1 &= \mathtt{seq}(setcdr(x, u_0), [\![\Phi]\!]). \end{aligned}$$

Furthermore, without loss of generality, assume that $\mathrm{FV}(\Sigma_i) \subseteq \mathrm{Dom}(\beta)$. We show that $\beta; \mu_0 \models_{\mathcal{L}} \Sigma_0$ iff $\beta; \mu_1 \models_{\mathcal{L}} \Sigma_1$. The result then follows by observing that $\beta; \mu_0 \models \Phi_0$ iff $\beta; \mu_1 \models \Phi_1$. Clearly $\beta; \mu_i \models_{\mathcal{L}} \{cdr(x) = u_i\}$ since by construction $cdr_{\mu_i}(c) = \beta(u_i)$. Thus it suffices to show that for any $\varphi \in \Sigma$, $\mathfrak{M}_{\mu_0} \models \varphi[\beta] \Leftrightarrow \mathfrak{M}_{\mu_1} \models \varphi[\beta]$. This is trivially true if φ is of the form $cell(y), \neg cell(y), u_0 = u_1, \neg(u_0 = u_1)$ or $car(y) = u$, so suppose that $(cdr(y) = u) \in \Sigma$. Since x is *cdr*-less in Σ we have that $\Sigma \models \neg(x = y)$, consequently $cdr_{\mu_0}(\beta(y)) = cdr_{\mu_1}(\beta(y))$. Thus $\mathfrak{M}_{\mu_0} \models (cdr(y) = u)[\beta]$ iff $\mathfrak{M}_{\mu_1} \models (cdr(y) = u)[\beta]$. $\square_{\mathbf{set}}$

$\square_{\mathbf{Soundness}}$

6. Completeness

In this section we state and prove the completeness theorem.

Theorem (Completeness): $\Sigma \models \Phi$ implies $\Sigma \vdash \Phi$ if there are no occurrences of function symbols $f \in \mathrm{F}_n$ in Φ.

The proof of the completeness theorem involves the symbolic evaluation of arbitrary expressions, with respect to a suitable set of constraints, to a canonical form. The symbolic evaluation of an expression, with respect to a set of constraints Σ, requires keeping track of three things: the newly allocated memory; the modifications to the original memory (described by Σ); and the remaining computation. The remainder of a computation is simply an expression. The newly allocated memory and the modifications to the original memory are represented by special kinds of contexts called syntactic memory contexts, Γ, and modification contexts, M, respectively. Using these contexts we define, relative to Σ, a form of syntactic reduction, $\overset{*}{\mapsto}_{\Sigma}$. It is defined in such a way that

$$(e \overset{*}{\mapsto}_{\Sigma} e') \Rightarrow (\Sigma \vdash e \simeq e').$$

Suppose $\Sigma \models e_0 \simeq e_1$, and there are no occurrences of function symbols $f \in \mathrm{F}_n$ in e_0, e_1. If Σ contains enough information concerning the nature of the free variables of e_i, then we can find $\Gamma_i; M_i; e'_i$

$$e_i \overset{*}{\mapsto}_{\Sigma} \Gamma_i; M_i; e'_i$$

and either $e'_i \in \{R[\![\vartheta(u_i)]\!], R[\![set\vartheta(u_i, u'_i)]\!]\}$, and $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u_i)$ or else $e'_i = u_i$. The former case corresponds to a *stuck state.* In the latter case the canonical form of e_i is then defined simply to be $\Gamma_i; M_i; u_i$. We show that one can use the introduction on the left rules to force Σ to contain the necessary information. Consequently suppose that Σ does contain sufficient information and that the canonical form of e_i is $\Gamma_i; M_i; u_i$. Then we have $\Sigma \vdash e_i \simeq \Gamma_i; M_i; u_i$. Thus by soundness $\Sigma \models e_i \simeq \Gamma_i; M_i; u_i$. Consequently $\Sigma \models \Gamma_0; M_0; u_0 \simeq \Gamma_1; M_1; u_1$. The completeness result then follows by showing that equivalent canonical forms are provably equivalent.

To obtain additional insight, consider the semantic question of deciding for any Σ and Φ whether $\Sigma \models \Phi$. Since all computations terminate we can decide for any $\beta; \mu$ such that $\mathrm{FV}(\Phi) \subseteq \mathrm{Dom}(\beta)$ whether $\beta; \mu \models \Phi$. The size or rank of an assertion is just the maximum of the sizes or ranks of its left-hand- and right-hand-expressions. The size or rank of an expression is just the usual notion. We say Σ is complete for Φ if Σ determines the structure of its models up to depth the size of Φ. If $\mathrm{FV}(e) \subseteq \mathrm{Dom}(\beta)$, the size of e is $\leq n$, and $\beta; \mu_0$ and $\beta; \mu_1$ are the same to depth n (agree on cells reachable from $\mathrm{Rng}(\beta)$ by paths of length $\leq n$), then $e; \beta; \mu_0$ and $e; \beta; \mu_1$ have the same computation sequences. Thus if Σ is complete for Φ, then to decide $\Sigma \models \Phi$ we need only pick some $\beta; \mu$ such that $\beta; \mu \models \Sigma$ and $\mathrm{FV}(\Phi) \subseteq \mathrm{Dom}(\beta)$ and check whether $\beta; \mu \models \Phi$ (For consistent Σ it is easy to find such models). Finally we note that for any Σ, Φ we can find a finite set of constraints $\{\Sigma_i \mid i < N\}$ such that

- for $i < N$, Σ_i is complete for Φ,
- for $i < N$, any model of Σ_i is a model of Σ, and
- any model of Σ is a model of Σ_i for some (unique) $i < N$.

Thus $\Sigma \models \Phi \Leftrightarrow (\forall i < N)(\Sigma_i \models \Phi)$ and we have seen how to decide the right hand side of the equivalence.

The completeness proof parallels the decidability argument using syntactic representations of memories and reduction. We begin by developing these representations. We then present the key lemmas for the proof of completeness and the proof itself. Finally we prove the lemmas.

6.1. Memory contexts and Modifications

Definition (Memory contexts): The syntactic analog of a memory is a memory context, Γ, which is a context of the form

$$\begin{array}{l}\mathtt{let}\{z_1 := \mathit{cons}(\mathtt{T},\mathtt{T})\} \ldots \mathtt{let}\{z_n := \mathit{cons}(\mathtt{T},\mathtt{T})\} \\ \quad \mathtt{seq}(\mathit{setcar}(z_1, u^a_1), \mathit{setcdr}(z_1, u^d_1), \ldots, \mathit{setcar}(z_n, u^a_n), \mathit{setcdr}(z_n, u^d_n), \varepsilon).\end{array}$$

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 $\mathit{car}_\Gamma, \mathit{cdr}_\Gamma \in [\mathrm{Dom}(\Gamma) \to \mathbb{U}]$ by $\mathit{car}_\Gamma(z_i) = u^a_i$ and $\mathit{cdr}_\Gamma(z_i) = u^d_i$. 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' such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$ and

$$\vartheta_{\Gamma'}(z') = \begin{cases} u_\vartheta & \text{if } z' = z \\ \vartheta_\Gamma(z') & \text{otherwise.} \end{cases}$$

$\Gamma\{car(z) = u\}$ is defined, for $z \in \mathrm{Dom}(\Gamma)$, to be the memory context Γ' such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma)$ and

$$car_{\Gamma'}(z') = \begin{cases} u & \text{if } z' = z \\ car_{\Gamma}(z') & \text{otherwise} \end{cases} \quad \text{and} \quad cdr_{\Gamma'}(z') = cdr_{\Gamma}(z').$$

Similarly for $\Gamma\{cdr(z) = u\}$. In the case when Γ is empty, $\Gamma = \varepsilon$, we write $\{z := [u_{car}, u_{cdr}]\}$ instead of $\varepsilon\{z := [u_{car}, u_{cdr}]\}$.

To express the constraints implicit in a memory context Γ we define for any Σ the extension of Σ by Γ relative to a given set of variables X.

Definition (Σ_Γ^X): If $X \in \mathbf{P}_\omega(\mathbb{X} - \mathrm{Dom}(\Gamma))$ and $\mathrm{FV}(\Sigma) \cap \mathrm{Dom}(\Gamma) = \emptyset$, then we define Σ_Γ^X as follows

$$\Sigma_\Gamma^X = \Sigma \cup \Delta_{\text{contents}} \cup \Delta_{\text{distinct}}$$

$$\Delta_{\text{contents}} = \bigcup_{z \in \mathrm{Dom}(\Gamma)} \{cell(z), \vartheta(z) = u_\vartheta \mid u_\vartheta = \vartheta_\Gamma(z)\}$$

$$\Delta_{\text{distinct}} = \bigcup_{z \in \mathrm{Dom}(\Gamma)} \{\neg(z = y) \mid y \in \mathrm{FV}(\Sigma) \cup X \cup (\mathrm{Dom}(\Gamma) - \{z\})\}.$$

The effects that the evaluation of an expression has on the original memory, described by constraints, are represented by contexts called modifications. They are simply sequences of assignments to variables that are not in the domain of the memory context, but are assumed to be cells.

Definition (Modifications): A *modification*, M, is a context of the form

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

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

6.2. Σ-Reduction

In analogy to the semantic reduction relations we define the relations $\xrightarrow{\mathrm{p}}_\Sigma$, $\mapsto_\Sigma$, 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 $\mathrm{Cells}(\Sigma)$, Σ decides equality on $\mathrm{Cells}(\Sigma)$, 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 $\mathrm{Cells}(\Sigma)$. (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.

Definition (Coherence): If Γ is a memory context and M is a modification as above then we say $(\Sigma, \Gamma; M)$ is coherent, written $Coh(\Sigma, \Gamma; M)$, if the following five conditions hold:

(1) $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(\Sigma) = \emptyset$

(2) $\mathrm{Dom}(M) \subseteq \mathrm{Cells}(\Sigma)$

(3) If $x_0, x_1 \in \mathrm{Dom}(\vartheta_M)$ are distinct, then $\Sigma \models \neg(x_0 = x_1)$.

(4) If $x_0, x_1 \in \mathrm{Cells}(\Sigma)$, then $\Sigma \models (x_0 = x_1)$ or $\Sigma \models \neg(x_0 = x_1)$.

(5) If $x \in \mathrm{Cells}(\Sigma)$, then there is at most one formula $(\vartheta(z) = u) \in \Sigma$ with $\Sigma \models (z = x)$, and if $(\vartheta(z) = u) \in \Sigma$, then $x \in \mathrm{Cells}(\Sigma)$.

We write $Coh(\Sigma, M)$ for $Coh(\Sigma, \Gamma; M)$ when $\mathrm{Dom}(\Gamma)$ is empty, when $\mathrm{Dom}(M)$ is empty we write $Coh(\Sigma, \Gamma)$ for $Coh(\Sigma, \Gamma; M)$, and when $\mathrm{Dom}(\Gamma)$ and $\mathrm{Dom}(M)$ are both empty we write $Coh(\Sigma)$ for $Coh(\Sigma, \Gamma; M)$.

Definition ($M\{\vartheta(z) = u\}_\Sigma$): Suppose that M is a modification, $Coh(\Sigma, M)$ and $z \in \mathrm{Cells}(\Sigma)$. Then $M\{car(z) = u\}_\Sigma$ is defined to be the modification M' with $\mathrm{Dom}(car_{M'}) = \mathrm{Dom}(car_M) \cup \{z\}$, $\mathrm{Dom}(cdr_{M'}) = \mathrm{Dom}(cdr_M)$, and for $z' \in \mathrm{Dom}(\vartheta_{M'})$

$$car_{M'}(z') = \begin{cases} car_M(z') & \text{if } \Sigma \models \neg(z = z') \\ u & \text{if } \Sigma \models (z = z') \end{cases} \quad \text{and} \quad cdr_{M'}(z') = cdr_M(z').$$

Similarly for $M\{cdr(z) = u\}_\Sigma$.

Definition ($\xrightarrow{\mathrm{p}}_\Sigma$): For Σ and $\Gamma; M$ such that $Coh(\Sigma, \Gamma; M)$ we define the relation $\Gamma; M[\![e]\!] \xrightarrow{\mathrm{p}}_\Sigma \Gamma'; M'[\![e']\!]$ as follows (letting $X = \mathrm{FV}(\Gamma; M[\![e]\!])$)

$$\Gamma; M[\![cell(u)]\!] \xrightarrow{\mathrm{p}}_\Sigma \begin{cases} \Gamma; M[\![\mathtt{Nil}]\!] & \text{if } \Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u) \\ \Gamma; M[\![\mathtt{T}]\!] & \text{if } \Sigma^X_\Gamma \models cell(u) \end{cases}$$

$$\Gamma; M[\![\vartheta(u)]\!] \xrightarrow{\mathrm{p}}_\Sigma \begin{cases} \Gamma; M[\![\vartheta_\Gamma(u)]\!] & \text{if } u \in \mathrm{Dom}(\Gamma) \\ \Gamma; M[\![\vartheta_M(u)]\!] & \text{if } (\exists u' \in \mathrm{Dom}(\vartheta_M))(\Sigma \models (u' = u)), \text{ or} \\ \Gamma; M[\![u']\!] & \text{if } u \in \mathrm{Cells}(\Sigma) \wedge \Sigma \models (\vartheta(u) = u') \end{cases}$$

$$\Gamma; M[\![eq(u_0, u_1)]\!] \xrightarrow{\mathrm{p}}_\Sigma \begin{cases} \Gamma; M[\![\mathtt{T}]\!] & \text{if } \Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models u_0 = u_1 \\ \Gamma; M[\![\mathtt{Nil}]\!] & \text{if } \Sigma^X_\Gamma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg(u_0 = u_1) \end{cases}$$

$$\Gamma; M[\![cons(u_0, u_1)]\!] \xrightarrow{\mathrm{p}}_\Sigma \Gamma\{z := [u_0, u_1]\}; M[\![z]\!] \quad \text{if } z \in \mathbb{X} - (\mathrm{Dom}(\Gamma) \cup \mathrm{FV}(\Sigma) \cup X)$$

$$\Gamma; M[\![set\vartheta(u, u')]\!] \xrightarrow{\mathrm{p}}_\Sigma \begin{cases} \Gamma\{\vartheta(u) = u'\}; M[\![u]\!] & \text{if } u \in \mathrm{Dom}(\Gamma) \\ \Gamma; M\{\vartheta(u) = u'\}_\Sigma[\![u]\!] & \text{if } u \in \mathrm{Cells}(\Sigma) \end{cases}$$

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.

Definition ($\mapsto_\Sigma$): For Σ and $\Gamma; M$ such that $Coh(\Sigma, \Gamma; M)$ we define the relation $\Gamma; M; R[\![e]\!] \mapsto_\Sigma \Gamma'; M'; R[\![e']\!]$ as follows. Let $X = \mathrm{FV}(\Gamma; M; R[\![e]\!])$. Then

$$\text{(if)} \qquad \Gamma; M; R[\![\mathtt{if}(u, e_1, e_2)]\!] \mapsto_\Sigma \begin{cases} \Gamma; M; R[\![e_1]\!] & \text{if } \Sigma_\Gamma^X \cup \mathrm{Diag}(\mathfrak{A}) \models \neg(u = \mathtt{Nil}) \\ \Gamma; M; R[\![e_2]\!] & \text{if } \Sigma \models (u = \mathtt{Nil}) \end{cases}$$

$$\text{(beta)} \qquad \Gamma; M; R[\![\mathtt{let}\{x := u\}e]\!] \mapsto_\Sigma \Gamma; M; R[\![e\{x := u\}]\!]$$

$$\text{(rec)} \qquad \Gamma; M; R[\![f(u_1, \ldots, u_n)]\!] \mapsto_\Sigma \Gamma; M; R[\![e\{x_1 := u_1, \ldots, x_n := u_n\}]\!]$$

$$\text{(delta)} \qquad \Gamma; M; R[\![\mathfrak{f}(u_1, \ldots, u_n)]\!] \mapsto_\Sigma \Gamma'; M'; R[\![u']\!]$$

where in (**rec**) we assume that $\langle f(x_1, \ldots, x_n) \leftarrow e \rangle$ is in δ and the x_i are chosen fresh, and in (**delta**) we assume that $\mathfrak{f} \in \mathbf{F}_n$, $\Gamma; M[\![\mathfrak{f}(u_1, \ldots, u_n)]\!] \xrightarrow{\mathrm{p}}_\Sigma \Gamma'; M'; u'$ and $\mathrm{Dom}(\Gamma') - \mathrm{Dom}(\Gamma)$ is disjoint from $\mathrm{FV}(\Gamma; M; R[\![\mathfrak{f}(u_1, \ldots, u_n)]\!])$.

Lemma (Coherence): Coherence is preserved by syntactic reduction.

If a modification, M, and a constraint set, Σ, are coherent, then the modification of Σ implicit in M is made explicit in Σ_M. To construct Σ_M from Σ we first remove the set of all assertions in Σ concerning components of cells that are mutated by M. The set removed is referred to as Δ_M^{forget}. Then we add to $\Sigma - \Delta_M^{\text{forget}}$ the set of assertions, Δ_M^{assign} concerning the components updated by M.

Definition (Σ_M): For $Coh(\Sigma, M)$ we define Σ_M as follows

$$\Sigma_M = (\Sigma - \Delta_M^{\text{forget}}) \cup \Delta_M^{\text{assign}}$$

$$\Delta_M^{\text{assign}} = \{\vartheta(z) = u_\vartheta \mid u_\vartheta = \vartheta_M(z), z \in \mathrm{Dom}(\vartheta_M)\}$$

$$\Delta_M^{\text{forget}} = \{(\vartheta(x) = u) \in \Sigma \mid (\exists z \in \mathrm{Dom}(\vartheta_M))(\Sigma \models x = z)\}$$

The Context Modification Introduction lemma combines and generalizes the *cons* and *set*ϑ introduction rules to arbitrary memory–modification contexts.

Lemma (CMI): If $Coh(\Sigma, \Gamma; M)$, $\Phi \in \mathbb{L}$, and $X = \mathrm{FV}(\Gamma; M; R[\![\Phi]\!])$ then

$$\frac{(\Sigma_\Gamma^X)_M \vdash \Phi}{\Sigma \vdash \Gamma; M; R[\![\Phi]\!]}$$

is derivable.

Proof (CMI): This is a simple consequence of the introduction rules (**cons.iii**) and (**set.vii**), together with the congruence rules and the definition of coherence (particularly the fifth condition). The only point to observe is that if Σ is the disjoint union of Σ' and $\{car(z_i) = w_i^a, cdr(z_i) = w_i^d\}_{z_i \in \mathrm{Dom}(M)}$, then each z_i is *car*-less and *cdr*-less in Σ'. $\Box_{\text{CMI}}$

6.3. Proof of Completeness

Before we state the key lemmas, we require one last set of definitions. As we mentioned earlier, syntactic reduction is defined so that if Σ contains enough information concerning the nature of the free variables of e, then

$$e \overset{*}{\mapsto}_{\Sigma} \Gamma; M; e',$$

and either $e' \in \{R[\![\vartheta(u)]\!], R[\![set\vartheta(u, u')]\!]\}$, and $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u)$ or else $e' = u$. The last case corresponds to the successful reduction of the expression to a value, while the former case corresponds to a *stuck state.*

Definition (Σ-stuck state): An expression e is said to reduce to a Σ-stuck state if $e \overset{*}{\mapsto}_{\Sigma} \Gamma; M[\![e']\!]$, and either $e' = R[\![\vartheta(u_0)]\!]$ or $e' = R[\![set\vartheta(u_0, u_1)]\!]$, and $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u_0)$.

In order to formalize the notion of a constraint set Σ containing enough information, we make the following definitions. $\mathrm{At}(X)$ is the set of atoms occurring in X. A *car-cdr* chain of length n is a reduction context of the form $\Theta = \vartheta_1(\vartheta_2(\ldots\vartheta_n(\varepsilon)\ldots))$ where $\vartheta_j \in \{car, cdr\}$. Note that the chain of length 0 is just ε. Finally we define the notion of n-completeness for constraint sets relative to a finite set of variables and atoms. The idea is that such a constraint set contains sufficient information to completely determine the evaluation of any expression of size less than n built from the given variables and atoms.

Definition (n-Complete w.r.t. $[\bar{x}, A]$): Σ is n-complete w.r.t. $[\bar{x}, A]$ if for every Θ, Θ_0, *car-cdr* chains of length $\leq n$, and $y, y_0 \in \bar{x}$, if $\Sigma \models \Theta[\![y]\!] = u$ and $\Sigma \models \Theta_0[\![y_0]\!] = u_0$, then

$$(\Sigma \models cell(u)) \vee (\Sigma \models \neg cell(u))$$

$$(\Sigma \models u = \alpha) \vee (\Sigma \models \neg(u = \alpha)) \quad \alpha \in A \cup \{\mathtt{T}, \mathtt{Nil}, u_0\}$$

$$(\Sigma \models cell(u)) \Rightarrow (\exists u_a, u_d \in \mathbb{U})((\Sigma \models car(u) = u_a) \wedge (\Sigma \models cdr(u) = u_d))$$

$$(\Sigma \models \neg cell(u)) \Rightarrow \neg(\exists u_a, u_d \in \mathbb{U})((\Sigma \models car(u) = u_a) \vee (\Sigma \models cdr(u) = u_d))$$

The following five lemmas enable a straightforward proof of the completeness theorem. Lemmas 0., 1., 3., and 4. hold for the full first-order language. Lemma 2. holds only for those expressions that contain no occurrences of recursively defined function symbols.

Lemma (0): If Σ is inconsistent, then $\Sigma \vdash \Phi$, for any $\Phi \in \mathbb{L}$.

Lemma (1): If $e \overset{*}{\mapsto}_{\Sigma} e'$, then $\Sigma \vdash e \simeq e'$.

Lemma (2): Assume e contains no occurrences of recursively defined function symbols. If Σ is $r(e)$-complete w.r.t. $[\mathrm{FV}(e), \mathrm{At}(\Sigma, e)]$ and $Coh(\Sigma)$, then either e reduces to a Σ-stuck state, or else there exists $\Gamma; M$, and a u such that $e \overset{*}{\mapsto}_{\Sigma} \Gamma; M[\![u]\!]$ and $Coh(\Sigma, \Gamma; M)$.

Lemma (3): For any consistent $\Sigma, \bar{x}$, $\Phi \in \mathbb{L}$, 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. $[\bar{x}, \mathrm{At}(\Sigma_i, \Phi)]$, and $Coh(\Sigma_i)$.
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 (4): 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 $\Sigma \models \Phi$, and Φ contains no occurrences of recursively defined function symbols. By lemma 0 we may assume that Σ is consistent. By lemma 3 it suffices to prove that $\Sigma \vdash \Phi$ under the added assumptions that $Coh(\Sigma)$ and Σ is $r(\Phi)$-complete w.r.t. $[\mathrm{FV}(\Phi), \mathrm{At}(\Sigma, \Phi)]$. By lemma 2 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 $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u_i)$.
2. $e'_i = R_i[\![set\vartheta_i(u_i, u'_i)]\!]$, $set\vartheta_i \in \{setcar, setcdr\}$ and $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u_i)$.
3. $e'_i = u_i$, and $Coh(\Sigma, \Gamma_i; M_i)$.

By lemma 1 we have $\Sigma \vdash e_i \simeq \Gamma_i; M_i[\![e'_i]\!]$, and by soundness we have $\Sigma \models e_i \simeq \Gamma_i; M_i[\![e']\!]$. We consider two cases, depending on the nature of Φ.

$(\Phi = \uparrow e)$ Since Σ is consistent, the case when $e' \in \mathbb{U}$ is impossible. In the other two cases we use (**S**), (**D**) and (**CMI**) to show that $\Sigma \vdash \uparrow \Gamma; M[\![e']\!]$, and hence that $\Sigma \vdash \uparrow e$.

$(\Phi = (e_0 \simeq e_1))$ 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 4 $\Sigma \vdash \Gamma_0; M_0[\![u_0]\!] \simeq \Gamma_1; M_1[\![u_1]\!]$.

$\square_{\textbf{Completeness}}$

6.4. Proofs of the Lemmas

Lemma (0): If Σ is inconsistent, then $\Sigma \vdash \Phi$, for any $\Phi \in \mathbb{L}$.

Proof (0): If Σ is inconsistent, then by (**Sat**) either $\Sigma \models \mathtt{T} = \mathtt{Nil}$ in the usual first-order interpretation, or else $\Sigma \models \neg cell(x)$ and $\Sigma \models \vartheta(x) = z$ for some $x, z \in \mathbb{U}$. In the former case the result follows by the structural rules and properties of `if`. In the later case it suffices to observe that $\Sigma \vdash \uparrow z$ and so since $\Sigma \vdash e\{y := z\} \simeq \mathtt{let}\{y := z\}e$ we can conclude, by choosing y new, that $\Sigma \vdash \uparrow e$ for any e. The result follows without much effort. $\square_0$

Lemma (1): If $e \overset{*}{\mapsto}_\Sigma e'$, then $\Sigma \vdash e \simeq e'$.

Proof (1): It suffices to show that if $Coh(\Sigma, \Gamma; M)$, then

$$\Gamma; M[\![e]\!] \mapsto_\Sigma \Gamma'; M'[\![e']\!] \Rightarrow (\Sigma \vdash \Gamma; M[\![e]\!] \simeq \Gamma'; M'[\![e']\!]).$$

Let $X = \mathrm{FV}(M[\![e]\!])$, $\Sigma' = (\Sigma_\Gamma^X)_M$, and note that the proof naturally divides up into three cases depending on the decomposition of e into $R[\![e_p]\!]$.

(if) In this case $e = R[\![\mathtt{if}(u, e_1, e_2)]\!]$ and by hypothesis either $\Sigma \models (u = \mathtt{Nil})$ or $\Sigma_\Gamma^X \cup \mathrm{Diag}(\mathfrak{A}) \models \neg(u = \mathtt{Nil})$. Thus either $\Sigma' \vdash u \simeq \mathtt{Nil}$ or $\Sigma' \vdash eq(u, \mathtt{Nil}) \simeq \mathtt{Nil}$, by **(S.i)**. In the former case $\Sigma' \vdash \mathtt{if}(u, e_1, e_2) \simeq e_2$ by **(if.ii,R.i,E)**, and so by **(CMI)**

$$\Sigma \vdash \Gamma; M; R[\![\mathtt{if}(u, e_1, e_2)]\!] \simeq \Gamma; M; R[\![e_2]\!].$$

In the latter case $\Sigma' \vdash \mathtt{if}(u, e_1, e_2) \simeq e_1$ by **(if.iii)** so again by **(CMI)**

$$\Sigma \vdash \Gamma; M; R[\![\mathtt{if}(u, e_1, e_2)]\!] \simeq \Gamma; M; R[\![e_1]\!].$$

(beta) In this case $e = R[\![\mathtt{let}\{x := u\}e_0]\!]$, and by **(let.ii)** $\Sigma' \vdash \mathtt{let}\{x := u\}e_0 \simeq e_0\{x := u\}$. Hence by **(CMI)**

$$\Sigma \vdash \Gamma; M; R[\![\mathtt{let}\{x := u\}e_0]\!] \simeq \Gamma; M; R[\![e_0\{x := u\}]\!].$$

(rec) In this case $e_p = f(\bar{u})$ and we use **(U, CMI, let.ii)**.

(delta) In this case $e_p = \mathfrak{f}(\bar{u})$ and consequently we may assume that

$$\Gamma; M[\![\mathfrak{f}(\bar{u})]\!] \xrightarrow{\mathrm{p}}_\Sigma \Gamma'; M'[\![u']\!]$$

and $(\mathrm{Dom}(\Gamma') - \mathrm{Dom}(\Gamma)) \cap \mathrm{FV}(\Gamma; M; R[\![\mathfrak{f}(\bar{u})]\!]) = \emptyset$. The proof naturally divides up into seven cases, depending on $\mathfrak{f}$. In four of these cases, corresponding to when $\mathfrak{f} \in \{cell, car, cdr, eq\}$, we have that $\Gamma = \Gamma'$ and $M = M'$. Consequently in these cases we need only show that $\Sigma' \vdash \mathfrak{f}(\bar{u}) \simeq u'$ and invoke **(CMI)** to obtain the result. We begin by considering these four cases.

($\mathfrak{f}(\bar{u}) = cell(u)$) In this case there are two possibilities, either $u' = \mathtt{Nil}$ or $u' = \mathtt{T}$. In the former case we have that $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u)$ and so $\Sigma' \cup \mathrm{Diag}(\mathfrak{A}) \models \neg cell(u)$. Hence by **(S.i)** we have that $\Sigma' \vdash \mathfrak{f}(\bar{u}) \simeq u'$. Similarly in the latter case we have that $\Sigma_\Gamma^X \models cell(u)$ and so $\Sigma' \models cell(u)$. Hence again by **(S.i)** we have that $\Sigma' \vdash \mathfrak{f}(\bar{u}) \simeq u'$.

($\mathfrak{f}(\bar{u}) = eq(u_0, u_1)$) Again there are two possibilities, either $u' = \mathtt{T}$ or $u' = \mathtt{Nil}$. In the former case we have that $\Sigma \cup \mathrm{Diag}(\mathfrak{A}) \models u_0 = u_1$ and so by construction of Σ' and **(S.i)** we have that $\Sigma' \vdash eq(u_0, u_1) \simeq \mathtt{T}$. In the case where $u' = \mathtt{Nil}$, we have that $\Sigma' \cup \mathrm{Diag}(\mathfrak{A}) \models u_0 \neq u_1$ and so by **(S.i)** $\Sigma' \vdash eq(u_0, u_1) \simeq \mathtt{Nil}$.

($\mathfrak{f}(\bar{u}) = car(u)$) In this case we have that $\Sigma' \models car(u) = u'$ and hence by **(S.i)** $\Sigma' \vdash car(u) \simeq u'$.

($\mathfrak{f}(\bar{u}) = cdr(u)$) This case is a trivial variation on *car*.

($\mathfrak{f}(\bar{u}) = \mathit{cons}(u_0, u_1)$) In this case we have that $\Gamma' = \Gamma\{u' := [u_0, u_1]\}$ and that $u' \notin \mathrm{Dom}(\Gamma) \cup X$. Now note that

$$\Sigma' \vdash \mathit{cons}(u_0, u_1) \simeq \mathit{setcdr}(\mathit{cons}(u_0, \mathtt{T}), u_1) \quad \text{by } (\mathbf{set.vi})$$

$$\simeq \mathit{setcdr}(\mathit{setcar}(\mathit{cons}(\mathtt{T}, \mathtt{T})), u_0), u_1) \quad \text{by } (\mathbf{set.v,R.i})$$

$$\simeq \mathtt{let}\{u' := \mathit{cons}(\mathtt{T}, \mathtt{T})\}\mathit{setcdr}(\mathit{setcar}(u', u_0), u_1) \quad \text{by } (\mathbf{let.i,R.iii})$$

$$\simeq \mathtt{let}\{u' := \mathit{cons}(\mathtt{T}, \mathtt{T})\}\mathit{setcdr}(\mathtt{seq}(\mathit{setcar}(u', u_0), u'), u_1)$$

by (**set.iii,R.i,CMI**)

$$\simeq \mathtt{let}\{u' := \mathit{cons}(\mathtt{T}, \mathtt{T})\}\mathtt{seq}(\mathit{setcar}(u', u_0), \mathit{setcdr}(u', u_1))$$

by (**R.ii,CMI**)

$$\simeq \mathtt{let}\{u' := \mathit{cons}(\mathtt{T}, \mathtt{T})\}\mathtt{seq}(\mathit{setcar}(u', u_0), \mathit{setcdr}(u', u_1), u')$$

by (**set.iii,CMI**)

Thus we have shown that $\Sigma' \vdash \mathit{cons}(u_0, u_1) \simeq \{u' := [u_0, u_1]\}; u'$, and so by (**CMI**)

$$\Sigma \vdash \Gamma; M; R[\![\mathit{cons}(u_0, u_1)]\!] \simeq \Gamma; M; R[\![\{u' := [u_0, u_1]\}; u']\!]$$

$$\simeq \Gamma; M; \{u' := [u_0, u_1]\}; R[\![u']\!] \quad \text{by } (\mathbf{Rii,Riii,CMI})$$

$$\simeq \Gamma\{u' := [u_0, u_1]\}; M; R[\![u']\!]$$

by (**Rii,Riii,cons.ii,cons.iii,CMI**) and (**Example.iii**)

($\mathfrak{f}(\bar{u}) = \mathit{setcar}(u_0, u_1)$) In this case $u_0 = u'$ and there are two possibilities, either $u_0 \in \mathrm{Cells}(\Sigma)$ or $u_0 \in \mathrm{Dom}(\Gamma)$. In the latter case, assuming that $\Gamma(u_0) = [u_0^a, u_0^d]$ we have that $\Gamma' = \Gamma\{u_0 := [u_1, u_0^d]\}$. Now by (**set.iii**) $\Sigma' \vdash \mathit{setcar}(u_0, u_1) \simeq \mathtt{seq}(\mathit{setcar}(u_0, u_1), u_0)$, and so by (**CMI**)

$$\Sigma \vdash \Gamma; M; R[\![\mathit{setcar}(u_0, u_1)]\!] \simeq \Gamma; M; R[\![\mathtt{seq}(\mathit{setcar}(u_0, u_1), u_0)]\!]$$

$$\simeq \Gamma; \mathtt{seq}(\mathit{setcar}(u_0, u_1), M; R[\![u_0]\!]) \quad \text{by } (\mathbf{S.i,Rii,set.i,set.iv,CMI})$$

$$\simeq \Gamma'; M; R[\![u']\!] \quad \text{by } (\mathbf{S.i,Rii,set.ii,set.iv,CMI})$$

while in the former case, assuming that $u_0 \in \mathrm{Dom}(M)$, we have that $M' = M\{\mathit{car}(u_0) = u_1\}$. Now by (**set.iii**) $\Sigma' \vdash \mathit{setcar}(u_0, u_1) \simeq \mathtt{seq}(\mathit{setcar}(u_0, u_1), u_0)$, and so by (**CMI**)

$$\Sigma \vdash \Gamma; M; R[\![\mathit{setcar}(u_0, u_1)]\!] \simeq \Gamma; M; R[\![\mathtt{seq}(\mathit{setcar}(u_0, u_1), u_0)]\!]$$

$$\simeq \Gamma; M; \mathtt{seq}(\mathit{setcar}(u_0, u_1), R[\![u_0]\!]) \quad \text{by } (\mathbf{S.i,Rii,set.i,set.iv,CMI})$$

$$\simeq \Gamma; M'; R[\![u']\!] \quad \text{by } (\mathbf{S.i,Rii,set.ii,set.iv,set.ii,CMI})$$

The case when $(\exists z)(\Sigma \models z = u_0 \wedge z \in \mathrm{Dom}(M))$ is almost identical to the above argument.

($\mathfrak{f}(\bar{u}) = setcdr(u_0, u_1)$) This case is a trivial variation on *setcar*.

$\square_1$

Lemma (2): Suppose that e does not contain any recursively defined function symbols. If Σ is $r(e)$-complete with respect to $[\mathrm{FV}(e), \mathrm{At}(\Sigma, e)]$, and $Coh(\Sigma)$, then either e reduces to a Σ-stuck state or else there exists $\Gamma; M$, and a u such that $e \overset{*}{\mapsto}_\Sigma \Gamma; M[\![u]\!]$ and $Coh(\Sigma, \Gamma; M)$.

Proof (2): This follows from the simple observation that if $e \mapsto_\Sigma e'$ and Σ is $r(e)$-complete w.r.t. $[\mathrm{FV}(e), \mathrm{At}(\Sigma, e)]$, then Σ is $r(e')$-complete w.r.t. $[\mathrm{FV}(e'), \mathrm{At}(\Sigma, e')]$. Consequently the three cases above are the only ones in which further reduction is not possible. $\square_2$

Lemma (3): For any consistent $\Sigma, \bar{x}$, $\Phi \in \mathbb{L}$, 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. $[\bar{x}, \mathrm{At}(\Sigma_i, \Phi)]$, and $Coh(\Sigma_i)$.
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.

Proof (3): This is a simple consequence of the introduction on the left rules. $\square_3$

Lemma (4): 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 (4): By lemma 0 we may assume that Σ is consistent. Using a simple construction from constants one can show that for any consistent Σ there is a $\beta; \mu$ such that

1. $\mathrm{Dom}(\beta) = \mathrm{FV}(\Sigma) \cup \mathrm{FV}(\Gamma_0; M_0[\![u_0]\!]) \cup \mathrm{FV}(\Gamma_1; M_1[\![u_1]\!])$ and $\beta; \mu \models_{\mathcal{L}} \Sigma$,
2. $\beta(x) = \beta(y)$ iff $\Sigma \models x = y$.

Given such a $\beta; \mu$ we show that if $e_0; \beta; \mu \simeq e_1; \beta; \mu$, then $\Sigma \vdash e_0 \simeq e_1$.

If $e_0; \beta; \mu \simeq e_1; \beta; \mu$, then there exists an object $v; \mu' \in \mathbb{O}$ with $\mathrm{Dom}(\mu) \subseteq \mathrm{Dom}(\mu')$, μ_0, μ_1 with $\mu' \subseteq \mu_i$, and $\beta_i \supseteq \beta$ with $\beta_i(u_i) = v$ such that $e_i; \beta; \mu \overset{*}{\mapsto} u_i; \beta_i; \mu_i$. Now put

$$G_i = \{x \in \mathrm{Dom}(\Gamma_i) \mid \beta_i(x) \in \mathrm{Dom}(\mu_i) - \mathrm{Dom}(\mu')\}$$

Then by construction $u_i \notin G_i$ and if $x \in \mathrm{Dom}(\Gamma_i) - G_i$, then $\vartheta_{\Gamma_i}(x) \notin G_i$. Similarly if $x \in \mathrm{Dom}(M_i)$, then $\vartheta_{M_i}(x) \notin G_i$. Consequently we can show that

$$\Sigma \vdash \Gamma_i; M_i[\![u_i]\!] \simeq \Gamma_{G_i}; \Gamma'_i; M_i[\![u_i]\!]$$

for Γ_{G_i} and Γ'_i memory contexts with the property that $\mathrm{Dom}(\Gamma_{G_i}) = G_i$ and

$$G_i \cap \mathrm{FV}(\Gamma'_i; M_i[\![u_i]\!]) = \emptyset.$$

Now by the garbage collection axioms we have that

$$\Sigma \vdash \Gamma_i; M_i[\![u_i]\!] \simeq \Gamma'_i; M_i[\![u_i]\!].$$

Also note that, putting $e'_i = \Gamma'_i; M_i[\![u_i]\!]$, that $e'_i; \beta; \mu \overset{*}{\mapsto} u_i; \beta_i; \mu'$. Consequently we can construct a bijection $\pi : \mathrm{Dom}(\Gamma'_0) \to \mathrm{Dom}(\Gamma'_1)$ such that (extending π as the identity off $\mathrm{Dom}(\Gamma'_0)$) $\Sigma \models \pi(\vartheta_{\Gamma'_0}(x)) = \vartheta_{\Gamma'_1}(\pi(x))$ for all $x \in \mathrm{Dom}(\Gamma'_0)$ $\Sigma \models \pi(u_0) = u_1$.

Consequently $\Gamma'_0; u_0$ and $\Gamma'_1; u_1$ differ only up to α-conversion and Σ-equality and hence we may assume they are the same. For $y \in \mathrm{Dom}(\vartheta_{M_0}) \cap \mathrm{Dom}(\vartheta_{M_1})$ we have $\Sigma \models \vartheta_{M_0}(y) = \vartheta_{M_1}(y)$ and we may assume they are the same. If $y \in \mathrm{Dom}(\vartheta_{M_0}) - \mathrm{Dom}(\vartheta_{M_1})$, then there must be some u such that $\Sigma \models \vartheta(y) = u$. Otherwise we could choose μ such that $\vartheta_\mu(\beta(y))$ is not the value assigned by M_0. In this case we can assume that $\vartheta_{M_0}(y) = u$, consequently for some M'_0 we have that $M_0 = M'_0; \mathtt{seq}(set\vartheta(y, u), \varepsilon)$. Using the derived rule (**Example.i**), $\{cell(x)\} \vdash setcar(x, car(x)) \simeq x$, we can prove

$$(\Sigma^X_{\Gamma_0})_{M'_0} \vdash set\vartheta(y, u) \simeq y$$

and hence that

$$\Sigma \vdash \Gamma_0; M'_0[\![u_0]\!] \simeq \Gamma_0; M_0[\![u_0]\!].$$

Consequently we can remove y from $\mathrm{Dom}(M_0)$. Repeating this we can transform M_0 and M_1 into the same modification. Hence $\Sigma \vdash e_0 \simeq e_1$.

$\Box_4$

7. Relating notions of equivalence and fragments

In Mason and Talcott [17, 19] we presented a study of operational equivalence and strong isomorphism in the presence of function abstractions and mutable binary cells. The first-order language presented in this paper can be thought of as a fragment of the higher-order language. Since both operational equivalence and strong isomorphism are relations defined relative to a class of contexts, it is of interest to compare these relations on various fragments. In this section we consider three fragments, zero-order, first-order, and full higher-order, and summarize results presented in [19] (where more detailed proofs may be found).

In order to distinguish analogous domains of different fragments we subscript the syntactic and semantic domain symbols by 'zo' for zero-order, 'fo' for first-order, and 'ho' for higher-order. Thus $\mathbb{E}_{\mathrm{fo}}$ is the set of first-order expressions (the set $\mathbb{E}$ defined in §2.) and $\mathbb{E}_{\mathrm{zo}}$, the zero-order expressions, is the set of first-order expressions that do not contain any function symbols $f \in \mathrm{F}$. $\mathbb{E}_{\mathrm{ho}}$ is the set of higher-order expressions, defined below.

Definition ($\mathbb{U}_{\text{ho}}$ $\mathbb{E}_{\text{ho}}$): The set of higher-order value expressions, $\mathbb{U}_{\text{ho}}$, and the set of higher-order expressions, $\mathbb{E}_{\text{ho}}$, are defined, mutually recursively, as follows:

$$\mathbb{U}_{\text{ho}} := \mathbb{X} + \mathbb{A} + \lambda\mathbb{X}.\mathbb{E}_{\text{ho}}$$
$$\mathbb{E}_{\text{ho}} := \mathbb{U} + \texttt{if}(\mathbb{E}_{\text{ho}}, \mathbb{E}_{\text{ho}}, \mathbb{E}_{\text{ho}}) + \texttt{app}(\mathbb{E}_{\text{ho}}, \mathbb{E}_{\text{ho}}) + \bigcup_{n\in\mathbb{N}} \mathbf{F}_n(\mathbb{E}^n_{\text{ho}})$$

The definitions of the various semantic domains can be found in [17]. An important point is that memories in $\mathbb{M}_{\text{ho}}$ can now contain closures (a lambda abstraction together with an environment from $\mathbb{B}_{\text{ho}}$). The notion of context, reduction context and redex is extended to take into account the enlarged syntax. The only modifications to the reduction relations, $\xrightarrow{\text{p}}$ and $\mapsto$, are to revise the (**equality**) clause, and to replace the (**rec**) clause by the (**beta-value**) clause. The new clauses are:

$$\text{(equality)} \qquad eq([v_0, v_1]; \mu) \xrightarrow{\text{p}} \begin{cases} \texttt{T}; \mu & \text{if } v_0 = v_1 \text{ and } v_i \in \mathbb{A} \cup \mathbb{C} \text{ for } i < 2 \\ \texttt{Nil}; \mu & \text{otherwise} \end{cases}$$

$$\text{(beta-value)} \quad R[\![\texttt{app}(u_0, u_1)]\!]; \beta; \mu \mapsto R[\![e_0]\!]; \beta \cup \beta_0\{x := \beta(u_1)\}; \mu$$

where in the (**beta-value**) clause we assume that $\beta(u_0) = \lambda x.e_0; \beta_0$, β and β_0 agree on the intersection of their domains, and $x \notin \text{Dom}(\beta \cup \beta_0)$.

In what follows $\Delta \in \{\text{zo}, \text{fo}, \text{ho}\}$, and ${}^{\varepsilon}\mathbb{E}_\Delta$ is the set of all contexts in the fragment $\mathbb{E}_\Delta$. We now define $\cong_\Delta$ to be operational equivalence with respect to the fragment $\mathbb{E}_\Delta$. Similarly we define $\simeq_\Delta$, to be strong isomorphism with respect to fragment $\mathbb{E}_\Delta$.

Definition ($\sqsubseteq_\Delta$ $\cong_\Delta$): For $e_0, e_1 \in \mathbb{E}_\Delta$ we define

$$e_0 \sqsubseteq_\Delta e_1 \;\Leftrightarrow\; (\forall C \in {}^{\varepsilon}\mathbb{E}_\Delta \mid \text{FV}(C[\![e_0]\!]) = \emptyset = \text{FV}(C[\![e_1]\!]))(\downarrow C[\![e_0]\!] \;\Rightarrow\; \downarrow C[\![e_1]\!])$$
$$e_0 \cong_\Delta e_1 \;\Leftrightarrow\; e_0 \sqsubseteq_\Delta e_1 \wedge e_1 \sqsubseteq_\Delta e_0$$

Definition ($\simeq$): Two expressions $e_0, e_1 \in \mathbb{E}_\Delta$ are strongly isomorphic if for every Δ-memory $\mu \in \mathbb{M}_\Delta$ and Δ-environment $\beta \in \mathbb{B}_\Delta$ such that $e_i; \beta; \mu \in \mathbb{D}_\Delta$ for $i < 2$ we have that one of the following holds:

1. $\uparrow e_1; \beta; \mu$ and $\uparrow e_2; \beta; \mu$, or
2. $(\exists v; \mu' \in \mathbb{O}_\Delta \mid \text{Dom}(\mu) \subseteq \text{Dom}(\mu'))(\bigwedge_{i<2}(\exists \mu_i \in \mathbb{M}_\Delta \mid \mu' \subseteq \mu_i)(e_i; \beta; \mu \rightarrow v; \mu_i))$

Note that first-order and zero-order value expressions coincide, and hence so do the respective notions of memory contexts and models. Also we will take $\cong_{\text{zo}}$ to be defined as a relation on first-order expressions (quantifying over zero-order contexts) as this simplifies the comparisons. The situation is summarized in the following theorem.

Theorem (Fragments):

$$\begin{tikzcd}
e_0 \cong_{\mathrm{ho}} e_1 \arrow[r, Rightarrow] \arrow[r, phantom, shift left=4ex, "\not\Leftarrow^{b}"] \arrow[d, Rightarrow, shift right, "\not"', "e"'] & e_0 \cong_{\mathrm{fo}} e_1 \arrow[r, Leftrightarrow] \arrow[d, Leftrightarrow, "c"] & e_0 \cong_{\mathrm{zo}} e_1 \arrow[d, Leftrightarrow, "d"] \\
e_0 \simeq_{\mathrm{ho}} e_1 \arrow[r, Rightarrow] \arrow[r, phantom, shift right=4ex, "\not\Leftarrow^{b}"] \arrow[u, Rightarrow, shift right, "a"'] & e_0 \simeq_{\mathrm{fo}} e_1 \arrow[r, Leftrightarrow] & e_0 \simeq_{\mathrm{zo}} e_1
\end{tikzcd}$$

Proof : The horizontal implications are simple consequences of the corresponding containment relations for the relevant contexts. The implication labeled (a) is a consequence of the weak extensionality property (**ciu**) that is proved in [19]. The negated implication (b) is due essentially to type discrimination capability of the language. A counterexample is $e_0 = eq(x, x)$ and $e_1 = \mathtt{T}$. Then we have $e_0 \cong_{\mathrm{fo}} e_1$ and $e_0 \simeq_{\mathrm{fo}} e_1$ but neither $e_0 \cong_{\mathrm{ho}} e_1$ nor $e_0 \simeq_{\mathrm{ho}} e_1$ hold since $eq(\lambda x.x, \lambda x.x) \simeq \mathtt{Nil}$.[1] The implication ($\Uparrow^c$) is a consequence of weak extensionality for the first-order fragment (**fo.ciu**), also proved in [19]. The implication ($\Downarrow^d$) follows from the fact that if $e_0 \simeq_{\mathrm{zo}} e_1$ does not hold then we can find a zero-order memory context Γ, a sequence of values $u_1, \ldots, u_n$, and reduction context R such that, letting C be $\Gamma[\![R[\![\mathtt{let}\{x_1 := u_1, \ldots, x_n := u_n\}\varepsilon]\!]]\!]$, $C[\![e_0]\!]$ is defined and $C[\![e_1]\!]$ is not defined. The converse arrows for (c,d) now follow from the above and the fact that the zero-order relations are meaningful for first-order expressions. An example that establishes the negated implication (e) is a simple matter. $\lambda x.x \cong_{\mathrm{ho}} \lambda x.\mathtt{seq}(x, x)$ but clearly the two value expressions are not strongly isomorphic. □

With the exception of the structural rules, the inference rules of our system (including induction) are sound in the higher-order case as well. The structural rules (actually the translation of constraints to assertions) must be modified to account for the fact that, as noted in the proof of (**Fragments**), equivalence of two value expressions does not imply their *eq*-ness. This is because computationally *eq* is not allowed to make any non-trivial distinctions between higher-order objects, while operational equivalence and strong isomorphism do make such distinctions. In fact, with the exception of (**examples.v**), all of the consequences given at the end of §3. lift to the higher-order case.

As noted in (**Fragments**) strong isomorphism is a stronger notion than operational equivalence for the full language, any two operationally equivalent λ-expressions will provide a counterexample, provided that they are distinct. What is surprising perhaps is that these are essentially the only counterexamples. The following theorem, a generalization of the theorem [14], p.48], states that operational equivalence and strong isomorphism coincide on a natural fragment of the full higher-order language, $\mathbb{E}_{\neg\lambda}$.

---

[1] This particular counterexample is an artifact of our choice of semantics for *eq*. However, any choice consistent with an extensional interpretation of operations has a corresponding counterexample.

Definition ($\mathbb{E}_{\neg\lambda}$): The set of λ-free expressions $\mathbb{E}_{\neg\lambda}$ is inductively defined as

$$\mathbb{A} + \mathbb{X} + \mathtt{app}(\mathbb{E}_{\neg\lambda}, \mathbb{E}_{\neg\lambda}) + \mathtt{if}(\mathbb{E}_{\neg\lambda}, \mathbb{E}_{\neg\lambda}, \mathbb{E}_{\neg\lambda}) + \mathtt{let}\{\mathbb{X} := \mathbb{E}_{\neg\lambda}\}\mathbb{E}_{\neg\lambda} + \bigcup_{n\in\mathbb{N}} \mathbf{F}_n(\mathbb{E}^n_{\neg\lambda})$$

Theorem (foc): If $e_0, e_1 \in \mathbb{E}_{\neg\lambda}$ and $e_0 \cong e_1$, then $e_0 \simeq e_1$.

Proof : See [19]. □

8. Summary and Conclusions

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 the zero-order fragment (programs that use only memory operations, and make no use of recursively defined functions, arithmetic operations, etc.). Thus the system can be seen to adequately express the semantics of memory operations. The system is also computationally adequate for the full first-order language, in the sense that any closed first-order expression that returns a value is provably equivalent to a canoniacl form. We have also indicated how induction principles can be added in order to reason about recursively defined functions. Presumably the completeness result could be extended to a relative completeness result for the first-order language and for extensions to abstract algebraic data types rather than unstructured atoms, but we have not explored this possibility.

Equivalence in all models (unconstrained equivalence) 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. With minor modification to the structural rules, the extended set of rules is also valid in the higher-order case, and provides a very useful tool for reasoning about program equivalence in the richer language.

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. Thus our work can be seen as an extension of the early work on Nelson and Oppen. There are three key algorithms in our procedure. 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. Oppen [26] gives a decision procedure for the first-order theory of pure Lisp, i.e. the theory of *cell*, *car*, *cdr*, *cons* over acyclic list structures. Nelsen and Oppen [25] treats the quantifier-free case over possibly cyclic list structures. Neither treats updating operations.

Work is in progress to extend the formal system to a full higher-order Scheme-like language (with untyped lambda abstraction). Felleisen [7, 8] gives an equational calculus for reasoning about Scheme-like programs, which is extended and simplified in Felleisen and Hieb [9]. 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. Moggi [22] shows that, in principle, purely equational reasoning in arbitrary computational monads can be lifted to higher-order intuitionistic logic. It is not clear just how the lifting construction distorts the reasoning, and further exploration of this approach is needed to determine if it can be used for proving properties of programs.

Acknowledgements

This research was partially supported by DARPA contract N00039-84-C-0211 and NSF grants CCR-8718605 and CCR-8917606 and CCR-8915663. 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, Jussi Ketonen, Elizabeth Wolf, and the three anonymous referees.

9. References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North Holland, Amsterdam, 1954.

[2] A. Avron, F. Honsell, and I. A. Mason. An overview of the Edinburgh Logical Framework. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification.* Springer Verlag, Heidelberg, 1989.

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

[4] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[5] C.C. Chang and H.J. Keisler. *Model Theory.* North Holland, Amsterdam, 1973.

[6] A. Demers and J. Donahue. Making variables abstract: An equational theory for Russell. In *10th ACM Symposium on Principles of Programming Languages*, 1983.

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

[8] M. Felleisen. λ-v-cs: An extended λ-calculus for Scheme,. In *1988 ACM conference on Lisp and functional programming*, volume 52, pages 72–85, 1988.

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, 1989.

[10] R. Harper, H. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science.* IEEE, 1987.

[11] J. M. Lucassen. *Types and Effects, Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT, 1987. Also available as LCS TR-408.

[12] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[13] I. A. Mason. Equivalence of first order Lisp programs: Proving properties of destructive programs via transformation. In *First Annual Symposium on Logic in Computer Science.* IEEE, 1986.

[14] I. A. Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[15] I. A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10, 1988.

[16] I. A. Mason and C. L. Talcott. Memories of S-expressions: Proving properties of Lisp-like programs that destructively alter memory. Technical Report STAN-CS-85-1057, Department of Computer Science, Stanford University, 1985.

[17] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[18] I. A. Mason and C. L. Talcott. Reasoning about programs with effects. In *Programming Language Implementation and Logic Programming, PLILP'90*, volume 456 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

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

[20] I. A. Mason and C. L. Talcott. Program transformation for configuring components. In *ACM/IFIP Symposium on Partial Evaluation and Semantics based Program Manipulation*, 1991.

[21] I. A. Mason and C. L. Talcott. Program transformation via constraint propagation, 1991. to appear.

[22] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

[23] J. H. Morris. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[24] P. Mosses. A basic abstract semantic algebra. In *Semantics of data types, international symposium, Sophia-Antipolis, June 1984, proceedings*, volume 173 of *Lecture Notes in Computer Science*. Springer, Berlin, 1984.

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

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

[27] G. Plotkin. Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1, 1975.