Programming, Transforming, and Proving with Function Abstractions and Memories

Ian Mason
Stanford University
IAM@SAIL.STANFORD.EDU

Carolyn Talcott
Stanford University
CLT@SAIL.STANFORD.EDU

1. Overview

Notions of program equivalence are fundamental to the process of program specification and transformation. Recent work of Talcott, Mason, and Felleisen establishes a basis for studying notions of program equivalence for programming languages with function and control abstractions operating on objects with memory. This work extends work of Landin, Reynolds, Morris and Plotkin. [Landin 1964] and [Reynolds 1972] describe high level abstract machines for defining language semantics. [Morris 1968] defines an extensional equivalence relation for the classical lambda calculus. [Plotkin 1975] extends these ideas to the call-by-value lambda calculus and defines the operational equivalence relation. Operational equivalence is the equivalence naturally associated with the operational approximation pre-ordering. One expression operationally approximates another if for all closing program contexts either the first expression is undefined or both expressions are defined and their values are indistinguishable (with respect to some primitive means of testing equality). [Talcott 1985], [Mason 1986], and [Talcott 1987] study operational approximation and equivalence for subsets of a language with function and control abstractions and objects with memory. [Felleisen 1987] defines reduction calculi extending the call-by-value lambda calculus to languages with control and assignment abstractions. Talcott, Mason, and Felleisen all apply their theories to expressing and proving properties of program constructs and of particular programs.

Reduction calculi and operational approximation both provide a sound basis for purely equational reasoning about programs. Calculi have the advantage that the reduction relations are inductively generated from primitive reductions (such as beta-conversion) by closure operations (such as transitive closure or congruence closure). Equations proved in a calculus continue to hold when the language is extended to treat additional language constructs. Operational approximation is, by definition, sensitive to the set of language constructs and basic data available. Using operational approximation we can express and prove properties such as non-termination, computation induction and existence of least fixed points which cannot even be expressed in reduction calculi. Studying the laws of operational approximation and discovering natural extensions to reduction calculi provide useful insight into the nature of program equivalence.

This paper presents a study of operational approximation and equivalence in the presence of function abstractions and objects with memory. In the remainder of this section we give an informal summary of our results illustrating properties of operational

approximation and equivalence and the effect of introducing objects with memory. In the second section we define the syntax and semantics of our language. In the third section three equivalent definitions of operational approximation and equivalence are given and basic laws are presented. In the fourth section we define a notion of recursion operator and give two examples. In the final section we discuss additional related work. In the full paper [Mason and Talcott 1989d] we develop methods of proving approximation and equivalence and use these methods to prove the results presented here.

Congruence: Operational approximation and equivalence are congruence relations on expressions and hence closed under substitution and abstraction.

Weak extensionality: One expression approximates another just if for every closed instantiation of every *use*, if the first expression is defined then so is the second. Here use means use of the value of an expression as an argument to a function which will also have access to the memory supplied by the instantiating context. This result corresponds to a weak form of extensionality and is the basis of several general techniques for proving operational approximation.

Evaluation: Operational equivalence is compatible with evaluation. If one expression evaluates to another then the two expressions are operationally equivalent. Equivalent expressions are either both undefined or evaluate to equivalent objects.

Strong isomorphism: The notion of strong isomorphism defined for the first-order fragment in [Mason 1986] lifts naturally to the higher-order case. Strong isomorpism implies operational equivalence and agrees with it on a natural subset of the higher-order language.

Non-extensionality: In the call-by-value lambda calculus two expressions are operationally approximate just if all closed instantiations of the expressions are operationally approximate. When objects with memory are added this form of extensionality fails.

Restricted eta: In the call-by-value lambda calculus we have $\lambda x.e(x) \cong e$ if x does not occur free in e and if e denotes a function. This is a restricted form of the eta-conversion rule. It is preserved when objects with memory are added, with the appropriate definition of "denotes a function".

Recursion operators: Recursion operators compute the least fixed point (with respect to operational approximation) of functionals and thus provide a mechanism for recursive definition. In a purely functional language recursion operators use self-application to implement recursion. When memory is introduced recursion operators may also use memory loops to implement recursion. All recursion operators are operationally equivalent on functionals.

2. Computation over Memory Structures.

In existing applicative languages there are two approaches to introducing objects with memory. We shall call these the Lisp approach and the ML approach. In the Lisp approach the semantics of lambda abstraction is modified so that upon application lambda variables are bound to newly allocated memory cells. Reference to a variable

returns the contents of the cell and there is an assignment operation (`setq` or `set!`) for updating the contents of the cell bound to a variable. In the ML approach cells are added as a data type and operations are provided for creating cells and for accessing and modifying their contents. Reference to the contents of a cell must be made explicit. In the Lisp approach one can no longer use beta-conversion to reason about program equivalence. Beta-conversion is not even meaningful in general, as variables that are assigned can not simply be replaced by values. For example the program $(\lambda x.\texttt{seq}(\texttt{setq}(x,1),x)2$ evaluates to 1. Replacing x by 2 in the body changes 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 it occurs in. To have a reasonable calculus one needs two sorts of variables: assignable and non-assignable. In the ML approach the semantics of lambda application is preserved and beta-value conversion remains a valid law for reasoning about programs. The Lisp approach provides a natural syntax since normally one wants to refer to the contents of a cell and not the cell itself. However the loss of the beta rule poses a serious problem for reasoning about programs. This approach also violates the principle of separating the mechanism for binding from that of memory allocation [Mosses 1984]. Following the Scheme tradition, [Felleisen 1987] takes the Lisp 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. There have been recent improvements in this calculus, but the problem of mixing binding and allocation is inherent in the approach.

We take the ML approach to introducing objects with memory, adding primitive operations that create, access, and modify memory cells to the call-by-value lambda calculus. We will work with S-expressions memories (memories with binary cells) as this is the natural extension of our work on the first-order case.

2.1. Syntax

We fix a countably infinite set of variables, $\mathbb{X}$, a countable set of atoms, $\mathbb{A}$, and a family of operation symbols $\mathbb{F} = \{\mathbb{F}_n \mid n \in \mathbb{N}\}$ ($\mathbb{F}_n$ is a set of n-ary operation symbols) with $\mathbb{X}$, $\mathbb{A}$, $\mathbb{F}_n$ for $n \in \mathbb{N}$ all pairwise disjoint. We assume $\mathbb{A}$ contains two distinct elements playing the role of booleans, `T` for *true* and `Nil` for *false*. From the given sets we define expressions, value expressions, contexts, and value substitutions.

Definition ($\mathbb{U}$ $\mathbb{E}$): The set of value expressions, $\mathbb{U}$, and the set of expressions, $\mathbb{E}$, are the least sets satisfying the following equations:

$$\mathbb{U} := \mathbb{X} + \mathbb{A} + \lambda\mathbb{X}.\mathbb{E}$$

$$\mathbb{E} := \mathbb{U} + \texttt{if}(\mathbb{E},\mathbb{E},\mathbb{E}) + \texttt{app}(\mathbb{E},\mathbb{E}) + \bigcup_{n\in\mathbb{N}} \mathbb{F}_n(\mathbb{E}^n)$$

We let $a, a_0, \ldots$ 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 $\mathbb{E}$. λ is a binding operator and free and bound variables of expressions are defined as usual. $\mathrm{FV}(e)$ is the set of free variables of e. Two expressions are considered equal if they are the same up to renaming of bound variables. $e\{x := e'\}$ is

the result of substituting e' for x in e taking care not to trap free variables of e'. $\mathbb{E}_X$ is the set of expressions whose free variables are among X. For example $\mathbb{E}_\emptyset$ is the set of closed expressions.

Definition (σ): A value substitution is a finite map σ from variables to value expressions. σ, σ_0, ... ranges over value substitutions. We write $\{x_i := u_i \mid i < n\}$ for the substitution σ with domain $\{x_i \mid i < n\}$ such that $\sigma(x_i) = u_i$ for $i < n$. e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$.

Definition (${}^\varepsilon\mathbb{E}$): Contexts are expressions with holes. We use ε to denote a hole. The set of contexts, ${}^\varepsilon\mathbb{E}$, is defined by

$${}^\varepsilon\mathbb{E} = \{\varepsilon\} + \mathbb{X} + \mathbb{A} + \lambda\mathbb{X}.{}^\varepsilon\mathbb{E} + \mathtt{if}({}^\varepsilon\mathbb{E}, {}^\varepsilon\mathbb{E}, {}^\varepsilon\mathbb{E}) + \mathtt{app}({}^\varepsilon\mathbb{E}, {}^\varepsilon\mathbb{E}) + \bigcup_{n\in\mathbb{N}} \mathbb{F}_n({}^\varepsilon\mathbb{E}^n)$$

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

In order to make programs easier to read we introduce some further abbreviations. Multi-ary application and abstraction is obtained by currying as usual and application is usually represented by juxtaposition rather than explicitly writing out `app`. `let` is lambda-application as usual. $\mathtt{seq}(e_0, \ldots, e_n)$ evaluates the expressions e_i in order, returning the value of the last expression. This can be represented using `let` or `if`.

2.2. Semantics

To define the operational semantics we fix a countable set of (names of) cells, $\mathbb{C}$, disjoint from $\mathbb{A}$, $\mathbb{X}$, and $\mathbb{F}$. The remaining semantic domains are pfns, values, environments, memories, memory objects and descriptions. Informally these domains are described as follows.

$\mathbb{P}$, the set of pfns (partial function descriptions), is the set obtained by closing a lambda expression in an environment whose domain contains the free variables of the lambda expression. $\mathbb{V}$, the set of values, consists of atoms, cells, and pfns. $\mathbb{B}$, the set of environments, is the set of finite functions from variables to values. $\mathbb{M}$, the set of memories, is the set 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. $\mathbb{O}^{(n)}$, the set of n-ary memory objects, is the set of pairs with first component an n-tuple of values and second component a memory, such that the cells in the n-tuple of values lie in the domain of memory. Elements of $\mathbb{O}^{(1)}$ are called objects, and we omit the superscript. $\mathbb{D}$, the set of memory object descriptions (or just descriptions), is the set of triples with first component an expression, second component an environment whose domain contains the free variables of the expression, and third component a memory such that any cell occurring in the range of the environment is in the domain of the memory.

We let c, c_0, ... range over $\mathbb{C}$, v, v_0, ... range over $\mathbb{V}$, β, β_0, ... range over $\mathbb{B}$, μ, μ_0, ... range over $\mathbb{M}$, $u;\mu$, $u_0;\mu_0$, ... range over $\mathbb{O}$, and $e;\beta;\mu$, $e_0;\beta_0;\mu_0$, ... range over $\mathbb{D}$. We use ";" as a tupling operation in some notations, for example in objects and descrip-

tions, since some components of the these tuples are also collections (sets or tuples) and we wish to emphasize the outer level tuple structure. We extend environments to act on value expressions whose free variables are in the domain of the environment by defining $\beta(a) = a$ for $a \in \mathbb{A}$ and $\beta(\lambda x.e) = \lambda x.e; \beta$, If β_0 and β_1 agree on the intersection of their domains then $\beta_0 \cup \beta_1$ is the environment with smallest domain extending both β_0 and β_1.

The basic semantic relations and their domains are:

Relation	Sign	Domain
Primitive evaluation	$\rightharpoonup$	$\bigcup_{n\in\mathbb{N}} \mathbb{O}^n \times \mathbb{O}$
Single-step	$\mapsto$	$\mathbb{D} \times \mathbb{D}$
Reduction	$\overset{*}{\mapsto}$	$\mathbb{D} \times \mathbb{D}$
Evaluation	$\hookrightarrow$	$\mathbb{D} \times \mathbb{O}$

Operations are partitioned into algebraic operations and memory operations. By algebraic operation we mean a function mapping $\mathbb{A}^n$ to $\mathbb{A}$ for some $n \in \mathbb{N}$. The action of memory operations is described by the primitive evaluation relation. Computation is a process of applying reductions to descriptions. $\mapsto$ is the single-step reduction relation on descriptions. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. The evaluation relation, $\hookrightarrow$, between descriptions and memory objects is reduction composed with the operation converting a value description $u; \beta; \mu$ into the corresponding memory object $\beta(u); \mu$.

The unary memory operations are $\{atom, cell, car, cdr\}$ and binary memory operations are $\{eq, cons, setcar, setcdr\}$. The remaining operations are assumed to be algebraic. The memory operations are interpreted relative to a given memory as follows. *atom* is the characteristic function – using the booleans `T` and `Nil` – of the atoms, *cell* is the characteristic function of the cells, and *eq* tests whether two values are identical. We call the pair of values assigned to a cell in a memory its components. *cons* takes two arguments, creates a new cell (extending the memory domain) with the pair of arguments as its components, 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 (c, v) the first of which must be a cell, *setcar* updates the given memory so that in the resulting memory the first component of c is v. *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.

Definition ($\rightharpoonup$): The primitive evaluation relation $\delta([v_0, \ldots, v_{n-1}]; \mu) \rightharpoonup v'; \mu'$ is the least relation satisfying the following conditions.

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

$$cell(v; \mu) \rightharpoonup \begin{cases} \texttt{T}; \mu & \text{if } v \in \mathbb{C} \\ \texttt{Nil}; \mu & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

where in the cases for $car, cdr, setcar$ and $setcdr$ we assume that $c \in \mathrm{Dom}(\mu)$ and $\mu(c) = [v_0, v_1]$.

Although formally *cons* is multi-valued, the values differ only by renaming of cells and we only distinguish them when necessary for bookkeeping purposes. 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.

The single-step relation is defined via a decomposition of a non-value expression into a *reduction context* and a *primitive expression*. A primitive expression is either the application of a pfn to a value (beta reduction), branching according to whether a test value is `Nil` or not, or the application of a primitive operation.

Definition ($\mathbb{E}_{\mathrm{prim}}$): The set of primitive expressions, $\mathbb{E}_{\mathrm{prim}}$, is defined as

$$\mathbb{E}_{\mathrm{prim}} = \mathtt{if}(\mathbb{U},\mathbb{E},\mathbb{E}) + \mathtt{app}(\mathbb{U},\mathbb{U}) + \bigcup_{n\in\mathbb{N}} \mathbb{F}_n(\mathbb{U}^n)$$

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

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

We let R, R' range over $\mathbb{R}$.

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 $e' \in \mathbb{E}_{\mathrm{prim}}$.

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

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

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

where in the (**beta**) 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 $x \notin \mathrm{Dom}(\beta\cup\beta_0)$. In the (**delta**) clause we assume $x \notin \mathrm{Dom}(\beta)$, $v_i = \beta(u_i)$ for $1 \le i \le n$, and either δ is an n-ary algebraic

operation, $v' = \delta(v_1, \ldots, v_n)$, and $v_1, \ldots, v_n \in \mathbb{A}^n$, or δ is an n-ary memory operation and $\delta([v_1, \ldots, v_n]; \mu) \rightharpoonup v'; \mu'$.

Definition ($\hookrightarrow \downarrow \uparrow$): A description $e; \beta; \mu \in \mathbb{D}$ evaluates to the object $v; \mu' \in \mathbb{O}$, (written $e; \beta; \mu \hookrightarrow v; \mu'$) if it reduces to a value description denoting that object. A description is defined (written $\downarrow e; \beta; \mu$) if it evaluates to some object and is undefined A description is undefined (written $\uparrow e; \beta; \mu$) if it is not defined.

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

$$\downarrow(e; \beta; \mu) \;\Leftrightarrow\; (\exists v; \mu')(e; \beta; \mu \hookrightarrow v; \mu')$$

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

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

2.3. Syntactic Interpretation of the Operational Semantics

We now show how to represent the semantic domains and computation purely in terms of syntactic objects. This is important for establishing a purely syntactic means of reasoning about program equivalence and as a tool for reasoning about memory objects and descriptions.

2.3.1. Representation of semantic domains

Value expressions are the syntactic analog of values. Value substitutions are the syntactic analog of environments. A special form of contexts called memory contexts are the syntactic analog of memories. The analog of a description is a memory context together with an expression and the analog of a memory object is a memory context together with a value expression.

Definition (Γ): A memory context Γ is a context of the form

$$\begin{aligned}&\texttt{let}\{z_1 := \mathit{cons}(\texttt{Nil}, \texttt{Nil})\} \ldots \texttt{let}\{z_n := \mathit{cons}(\texttt{Nil}, \texttt{Nil})\} \\ &\quad \texttt{seq}(\mathit{setcar}(z_1, u_1^a), \mathit{setcdr}(z_1, u_1^d), \ldots, \mathit{setcar}(z_n, u_n^a), \mathit{setcdr}(z_n, u_n^d), \varepsilon)\end{aligned}$$

where $z_i \neq z_j$ when $i \neq j$. We abbreviate Γ by $\{z_i := [u_i^a, u_i^d] \mid 1 \leq i \leq n\}$.

In analogy to semantic memories, for Γ as above we define the domain of Γ to be $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$ and $\Gamma(z_i) = [u_i^a, u_i^d]$ for $1 \leq i \leq n$. Two memory contexts are considered the same if they have the same domain and range. $\Gamma\{z := [u_a, u_d]\}$ is defined to be the memory context Γ' such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$ and

$$\Gamma'(z') = \begin{cases} [u_a, u_d] & \text{if } z' = z \\ \Gamma(z') & \text{otherwise.} \end{cases}$$

If Γ_0 and Γ_1 agree on the intersection of their domains the $\Gamma_0 \cup \Gamma_1$ is the memory context

Γ' with domain $\mathrm{Dom}(\Gamma_0) \cup \mathrm{Dom}(\Gamma_1)$ such that

$$\Gamma'(z) = \begin{cases} \Gamma_0(z) & \text{if } z \in \mathrm{Dom}(\Gamma_0) \\ \Gamma_1(z) & \text{if } z \in \mathrm{Dom}(\Gamma_1) \end{cases}$$

Definition $(\Gamma; e)$**:** A syntactic description is a pair with first component a memory context and second component an arbitrary expression. We do not require that the free variables of the expression be contained in the domain of the memory context. If the expression is a value expression then the description is also a syntactic memory object. $\Gamma; e$, $\Gamma_0; e_0$, ... range over syntactic descriptions.

2.3.2. Representing Computation

We define single-step reduction on syntactic descriptions as follows.

Definition $(\mapsto)$**:**

(beta) $\quad \Gamma; R[\![\mathtt{app}(\lambda x.e, u)]\!] \mapsto \Gamma; R[\![e\{x := u\}]\!]$

(if) $\quad \Gamma; R[\![\mathtt{if}(u, e_1, e_2)]\!] \mapsto \begin{cases} \Gamma; R[\![e_1]\!] & \text{if } u \in (\mathbb{A} - \{\mathtt{Nil}\}) \cup \mathbb{L} \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![e_2]\!] & \text{if } u = \mathtt{Nil} \end{cases}$

(delta) $\quad \Gamma; R[\![\delta(u_1, \ldots, u_n)]\!] \mapsto \Gamma'; R[\![u']\!]$

where in (**delta**) we assume that either δ is an n-ary algebraic operation, $u_1, \ldots, u_n \in \mathbb{A}^n$, $\delta(u_1, \ldots, u_n) = u'$, and $\Gamma = \Gamma'$ or $\Gamma; R[\![\delta(u_1, \ldots, u_n)]\!] \rightharpoonup \Gamma'; R[\![u']\!]$ where

$$\Gamma; R[\![atom(u)]\!] \rightharpoonup \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } u \in \mathbb{A} \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{if } u \in \mathbb{L} \cup \mathrm{Dom}(\Gamma) \end{cases}$$

$$\Gamma; R[\![cell(u)]\!] \rightharpoonup \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } u \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{if } u \in \mathbb{L} \cup \mathbb{A} \end{cases}$$

$$\Gamma; R[\![eq(u_0, u_1)]\!] \rightharpoonup \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } u_0 = u_1 \text{ and } u_0, u_1 \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{if } u_0 \neq u_1 \text{ and } \bigvee_{i<2} u_i \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{if } u_0 \neq u_1 \text{ and } \bigwedge_{i<2} u_i \in \mathbb{A} \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{if } \bigvee_{i<2} u_i \in \mathbb{L} \end{cases}$$

$$\Gamma; R[\![cons(u_0, u_1)]\!] \rightharpoonup \Gamma\{z := [u_0, u_1]\}; R[\![z]\!]$$

$$\Gamma; R[\![car(z)]\!] \rightharpoonup \Gamma; R[\![u_a]\!]$$

$$\Gamma; R[\![cdr(z)]\!] \rightharpoonup \Gamma; R[\![u_d]\!]$$

$$\Gamma; R[\![setcar(z, u)]\!] \rightharpoonup \Gamma\{z := [u, u_d]\}; R[\![z]\!]$$

$$\Gamma; R[\![setcdr(z, u)]\!] \rightharpoonup \Gamma\{z := [u_a, u]\}; R[\![z]\!]$$

where in the *cons* rule $z \notin (\mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R[\![u_i]\!])$, $i \leq 2$, and in the *car*, *cdr*, *setcar*, and *setcdr* rules we assume $z \in \mathrm{Dom}(\Gamma)$ and $\Gamma(z) = [u_a, u_d]$.

For any injection ξ from cells to variables such that $\mathbb{X} - \mathrm{Rng}(\xi)$ is countably infinite

there is a natural extension to the remaining semantic domains:

$$\xi(a) = a$$
$$\xi(e;\beta) = e^{\xi(\beta)}$$
$$\xi(\beta) = \{x := \xi(\beta(x)) \mid x \in \mathrm{Dom}(\beta)\}$$
$$\xi([v_1,\ldots,v_n]) = [\xi(v_1),\ldots,\xi(v_n)]$$
$$\xi(\mu) = \{\xi(c) := \xi(\mu(c)) \mid c \in \mathrm{Dom}(\mu)\}$$
$$\xi(v;\mu) = \xi(\mu);\xi(v)$$
$$\xi(e;\beta;\mu) = \xi(\mu);\xi(e;\beta)$$

We assume a fixed cell naming map ξ and write Γ_μ for $\xi(\mu)$ and e^β for $\xi(e;\beta)$.

Syntactic computation corresponds stepwise to semantic computation. Thus semantic entities that have the same syntactic representation are computationally indistinguishable. This is made precise in the following theorem.

Theorem (simulation):

(i) If $e_0;\beta_0;\mu_0 \mapsto e_1;\beta_1;\mu_1$, then $\Gamma_{\mu_0};e_0^{\beta_0} \mapsto \Gamma_{\mu_1};e_1^{\beta_1}$.

(ii) If $\Gamma_{\mu_0};e_0^{\beta_0} \mapsto \Gamma_2;e_2$, then we can find $e_1;\beta_1;\mu_1$ such that $\Gamma_{\mu_1};e_1^{\beta_1} = \Gamma_2;e_2$ and $e_0;\beta_0;\mu_0 \mapsto e_1;\beta_1;\mu_1$.

3. Operational Approximation and Equivalence

In this section we define the operational approximation and equivalence relations and study their general properties.

Definition ($\sqsubseteq$ $\cong$): Two expressions are operationally approximate, written $e_0 \sqsubseteq e_1$, if for any closing context E, if $E[\![e_0]\!]$ is defined then $E[\![e_1]\!]$ is defined. Two expressions are operationally equivalent, written $e_0 \cong e_1$, if they approximate one another.

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall E \in {}^{\varepsilon}\mathbb{E} \mid E[\![e_0]\!], E[\![e_1]\!] \in \mathbb{E}_\emptyset)(\downarrow E[\![e_0]\!] \Rightarrow \downarrow E[\![e_1]\!])$$
$$e_0 \cong e_1 \;\Leftrightarrow\; e_0 \sqsubseteq e_1 \wedge e_1 \sqsubseteq e_0$$

By definition operational approximation (and hence operational equivalence) is a congruence relation on expressions. However it is not necessarily the case that instantiations of equivalent expressions are equivalent even if the instantiation is defined. Note that `T` and `Nil` are not operationally equivalent. These observations are summarized in the following lemma.

Lemma (Congruence):

1. $e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall E \in {}^{\varepsilon}\mathbb{E})(E[\![e_0]\!] \sqsubseteq E[\![e_1]\!])$
2. $\downarrow e$ and $e_0 \cong e_1$ does not imply $e_0\{x := e\} \cong e_1\{x := e\}$.

3. $\neg(\mathtt{T} \cong \mathtt{Nil})$

Proof (congruence):

Case 1: Trivial.

Case 2: As a counter-example we have $eq(x,x) \cong \mathtt{T}$ but $eq(cons(\mathtt{T},\mathtt{T}), cons(\mathtt{T},\mathtt{T})) \cong \mathtt{Nil}$.

Case 3: The context $\mathtt{if}(\varepsilon, car(\mathtt{T}), \mathtt{T})$ will distinguish $\mathtt{T}$ and $\mathtt{Nil}$. $\square_{\mathbf{congruence}}$

An alternate definition of operational approximation and equivalence in the presence of basic data is the following. Define two closed expressions to be trivially approximate if whenever the first is defined then both return the same atom or both return cells, or both return pfns. Then define two expressions to be operationally approximate just if they are trivially approximate in all closing contexts. This is definition given by Plotkin. Both definitions are equivalent in this setting since equality on basic data is computable.

3.1. Weak extensionality

Another characterization of operational approximation and equivalence is obtained by extending the semantic characterizion of the maximum approximation relation given in [Talcott 1985]. Two expressions are approximate just if all closed instantiations are trivially approximate in all reduction contexts. Suitably generalized, this characterization remains valid in the presence of memory. We define the relation $\sqsubseteq^{ciu}$ — all closed instantiations of all uses are approximate — and show this to be the same as operational approximation. The $\sqsubseteq^{ciu}$ characterization of operational approximation is the key for proving many laws of approximation and equivalence.

Definition (ciu):

$$e_0 \sqsubseteq^{ciu} e_1 \;\Leftrightarrow\; (\forall \Gamma, \sigma, R \;\big|\; (\forall j < 2)(\Gamma[\![R[\![e_j^\sigma]\!]]\!] \in \mathbb{E}_\emptyset))(\downarrow(\Gamma[\![R[\![e_0^\sigma]\!]]\!]) \Rightarrow \downarrow(\Gamma[\![R[\![e_1^\sigma]\!]]\!]))$$

Theorem (ciu): $e_0 \sqsubseteq e_1 \;\Leftrightarrow\; e_0 \sqsubseteq^{ciu} e_1$

A direct corollary of the ciu characterization of operational approximation is the following weak form of extensionality.

Corollary (wk.ext):

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall \Gamma, \sigma, R \;\big|\; (\forall j < 2)(\Gamma[\![R[\![e_j^\sigma]\!]]\!] \in \mathbb{E}_\emptyset))(\Gamma[\![R[\![e_0^\sigma]\!]]\!] \sqsubseteq \Gamma[\![R[\![e_1^\sigma]\!]]\!])$$

A simple consequence of (**ciu**) is the fact that in the case of closed expressions we need only check definedness in all closed reduction contexts in order to verify operational approximation.

Theorem (op.closed): If $e_0, e_1 \in \mathbb{E}_\emptyset$ then

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall R \in \mathbb{R}_\emptyset)(\downarrow(R[\![e_0]\!]) \Rightarrow \downarrow(R[\![e_1]\!]))$$

In the absence of memory operations, two expressions are operationally approximate just if all closed instantiations of variables to values are approximate [Talcott 1985]. This property fails when objects with memory are introduced. The notion of all closed instantiations being approximate, as well as the result just mentioned, is made explicit in the following.

Definition ($\sqsubseteq^{ci}$):

$$e_0 \sqsubseteq^{ci} e_1 \;\Leftrightarrow\; (\forall \Gamma, \sigma \mid (\forall j < 2)(\Gamma[\![e_j^\sigma]\!] \in \mathbb{E}_\emptyset))(\Gamma[\![e_0^\sigma]\!] \sqsubseteq \Gamma[\![e_1^\sigma]\!])$$

Lemma (non.ext): $e_0 \sqsubseteq^{ci} e_1$ does not imply $e_0 \sqsubseteq e_1$,

Proof (non.ext): A counter example is $e_0 = \mathtt{seq}(setcar(c, \lambda x.x), \lambda x.x)$ and $e_1 = \mathtt{seq}(setcar(c, \lambda x.x), \lambda x.car(c)(x))$. $\square_{\mathbf{non.ext}}$

3.2. Strong isomorphism

In [Mason 1986], the notion of strong isomorphism was defined for the first-order subset of our language and a powerful collection of tools was developed for reasoning about this relation. Two expressions e_0 and e_1 are strongly isomorphic if for every closed instantiation either both are undefined are both are defined and evaluate to objects that are equal modulo the production of garbage.

Definition ($\simeq$): Two expressions are strongly isomorphic, written $e_0 \simeq e_1$, if for each Γ, σ such that $\Gamma[\![e_j^\sigma]\!] \in \mathbb{E}_\emptyset$ for $j < 2$ one of the following holds:

(1) $\uparrow(\Gamma; e_0^\sigma)$ and $\uparrow(\Gamma; e_1^\sigma)$, or

(2) there exists u, Γ', Γ_0, Γ_1 such that $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$, $\Gamma'[\![u]\!] \in \mathbb{E}_\emptyset$, $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_j) = \emptyset$ and $\Gamma; e_j^\sigma \overset{*}{\mapsto} (\Gamma_j \cup \Gamma'); u$ for $j < 2$.

A consequence of (**ciu**) is that strong isomorphism implies operational equivalence.

Theorem (striso): If $e_0 \simeq e_1$ then $e_0 \cong e_1$.

A corollary of (**striso**) is that operational equivalence is preserved by evaluation.

Corollary (eval): $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; \Gamma[\![e]\!] \cong \Gamma'[\![e']\!]$.

The following is a collection of laws of strong isomorphism, and by (**striso**) they are also laws of operational equivalence. They correspond to the context independent subset of a complete set of rules for reasoning about memory operations in a first-order setting [Mason and Talcott 1989b,c].

Corollary (laws):

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

(ii) $e \simeq \mathtt{let}\{x := e\}x$

(iii) $R[\![\mathtt{let}\{x := e_0\}e_1]\!] \simeq \mathtt{let}\{x := e_0\}R[\![e_1]\!]$ for x not free in R

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

(v) $\mathtt{if}(e_0, e_1, e_1) \simeq \mathtt{let}\{x := e_0\}e_1 \qquad x \notin \mathrm{FV}(e_1)$

(vi) $\mathtt{let}\{x_0 := cons(u_0, u_1)\}\mathtt{let}\{x_1 := e_0\}e \simeq \mathtt{let}\{x_1 := e_0\}\mathtt{let}\{x_0 := cons(u_0, u_1)\}e$
if x_0 not free in e_0 and x_1 not free in u_0, u_1

(vii) $\mathtt{seq}(setcar(x, y_0), setcar(x, y_1)) \simeq setcar(x, y_1)$

(viii) $\mathtt{seq}(setcar(x, y), x) \simeq setcar(x, y)$

(ix) $\mathtt{seq}(setcdr(x_0, x_1), setcar(x_2, x_3), e) \simeq \mathtt{seq}(setcar(x_2, x_3), setcdr(x_0, x_1), e)$

(x) $setcar(cons(z, y), x) \simeq cons(x, y) \simeq setcdr(cons(x, z), y)$

Corollary (gc): If Γ is memory context such that $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(e) = \emptyset$ then $\Gamma[\![e]\!] \cong e$.

The following theorem, a generalization of [Mason 1986], states that operational equivalence and strong isomorphism coincide on a natural fragment.

Definition ($\mathbb{E}_{\mathrm{fo}}$): The set of first order expressions $\mathbb{E}_{\mathrm{fo}}$ is inductively defined as

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

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

The eta rule for the pure lambda calculus has the form $e \cong \lambda x.e(x)$ if x is not free in e. In an appied calculus where there are objects that are not functions we need the additional restriction that e must denote a function. In the presence of memory objects, if we interpret "e denotes a function" as $e \cong \Gamma[\![\rho]\!]$ for some memory context Γ and some lambda abstraction $\rho = \lambda y.e'$ then the eta rule is not valid. If we interpret "e denotes a function" as $e \cong \rho$, then the eta rule is valid.

Lemma (non.eta): In general $\lambda x.(\Gamma[\![\lambda x.e]\!])x$ is not operationally equivalent to $\Gamma[\![\lambda x.e]\!]$.

Proof (non.eta): As a counter-example we have

$$\{z := [\mathtt{T}, \mathtt{Nil}]\}; \lambda x.\mathtt{let}\{y := car(z)\}\mathtt{seq}(setcar(z, x), y).$$

□non.eta

Lemma (eta): If $e \cong \lambda x.e'$ then $\lambda x.e(x) \cong e$.

4. Recursion Pfns

In [Talcott 1985] the notion of recursion operator was introduced. Recursion operators compute the least fixed point (with respect to operational approximation) of functionals and thus provide a mechanism for definition by recursion. The definition of recursion operator identifies the essential properties needed to prove the least-fixed-point property. In order to extend the recursion theorem to the world of memories and to permit recursion operators that make use of memory, we need to define the

analog of functional. There are two possibilities: (i) as for the non-memory case functionals are expressions of the form $\lambda f, x.e$ or (ii) a functional is memory object of the form $\Gamma[\![\lambda f, x.e]\!]$. In case (ii) letting $\varphi = \lambda f, x.e$ we have $rec(\Gamma[\![\varphi]\!]) \cong \Gamma[\![rec(\varphi)]\!]$ and $\Gamma[\![\varphi]\!](rec(\Gamma[\![\varphi]\!])) \cong \Gamma \cup \Gamma'[\![\varphi(rec(\varphi'))]\!]$ where $\Gamma'[\![\varphi']\!] = \Gamma[\![\varphi]\!]$ and $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma) = \emptyset$. So, in general $rec(\Gamma[\![\varphi]\!])$ and $\Gamma[\![\varphi]\!](rec(\Gamma[\![\varphi]\!]))$ will not be equivalent. Thus the meaningful object to take a fixed point of is the simple functional with no local memory.

Definition (recnop): A closed lambda expression rec is a recursion operator if there exists Γ, $\rho \in \mathbb{L}$, $p \in \mathbb{X} - \mathrm{Dom}(\Gamma)$, such that $\Gamma[\![\rho]\!] \in \mathbb{E}_{\{p\}}$ and the following two conditions hold:

(i) $rec(p) \overset{*}{\mapsto} \Gamma; \rho$

(ii) If $\varphi = \lambda f, x.e$ with $\mathrm{FV}(\varphi) \cap \mathrm{Dom}(\Gamma) = \emptyset$ then $\Gamma_\varphi; \rho_\varphi(x) \overset{*}{\mapsto} \Gamma_\varphi; e\{f := \rho_\varphi\}$ where $\Gamma_\varphi; \rho_\varphi = \Gamma\{p := \varphi\}; \rho\{p := \varphi\}$.

We call $\Gamma; \rho$ the associated fixed-point template for rec (with parameter p). Condition (i) says that $rec(\varphi)$ evaluates to $\Gamma_\varphi; \rho_\varphi$ uniformly in the functional parameter. Condition (ii) says that applying ρ_φ to any value in a memory context whose restriction to $\mathrm{Dom}(\Gamma)$ is Γ_φ reduces, without modifying memory, to a computation of the body of the functional e with f replaced by ρ_φ.

Although the functionals we compute fixed points of have no local memory, the fixed points themselves will in general be pfn objects that have local memory. Thus the least-fixed-point property is formulated in terms of pfn objects.

Theorem (recn): If rec and rec' are recursion operators then rec computes the least fixed-point of functionals and is operationally equivalent to rec' on functionals. For any functional φ and any pfn object ψ

(fix) $rec(\varphi) \cong \varphi(rec(\varphi))$

(min) $\varphi(\psi) \sqsubseteq \psi \;\Rightarrow\; rec(\varphi) \sqsubseteq \psi$

(eq) $rec(\varphi) \cong rec'(\varphi)$

Two examples of recursion operators are rec_v and rec_m. rec_v is a conventional call-by-value fixed-point combinator which uses self-application to create the recursive self-reference. rec_m is a recursion operator which uses the ability to create and update cells to create the necessary self-reference. The method is essentially identical to the one suggested in [Landin, 1964].

Definition (rec_v rec_m): rec_v and rec_m are defined by

$$rec_v = \lambda p.\mathtt{let}\{r := \lambda h.\lambda x.p(h(h), x)\}r(r)$$

$$rec_m = \lambda p.\mathtt{let}\{z := cons(\mathtt{T}, \mathtt{T})\}\mathtt{seq}(setcar(z, \lambda x.p(\lambda x.car(z)(x), x), \lambda x.car(z)(x))$$

Lemma (rec): rec_v and rec_m are recursion operators.

Proof (rec): The fixed-point templates for rec_v and rec_m are $\emptyset$; $\lambda x.p(\pi_p(\pi_p), x)$ where

$\pi_p = \lambda h.\lambda x.p(h(h),x)$, and $\{z := [\lambda y.p(\lambda x.car(z)(x),y),\mathtt{T}]\}; \lambda x.car(z)(x)$. $\square_{\mathbf{rec}}$

5. Conclusions

The results presented in this paper provide basic tools for specifying and reasoning about objects with memory and programs acting on such objects. Our language is close to existing applicative languages such as Lisp, Scheme, and ML. Memory can be represented as syntactic contexts. This simplifies the expression of many properties since it provides natural notions of parameterized memory objects, of binding, and of substitution for parameters. In addition the syntactic representation allows us to compute with open expressions and provides a natural scoping mechanism for memory simply using laws for bound variables. Many of the basic equivalence relations on memories and other semantic entities translate naturally into simple syntactic equivalences such as alpha-equivalence.

A key result is the (**ciu**) characterization of operational approximation and equivalence. This is the basis of several important methods for proving approximation and equivalence. (**ciu**) extends the **safety** theorem of [Felleisen 1987, thm 5.27, p.149]. Two expressions are safely equivalent if every closed instantiation of every use is provably equivalent in the assignment calculus. Since calculi can not express non-termination we have that safe equivalence implies operational equivalence but not conversely.

[Mason and Talcott 1989a] contains sample applications of our results. The first application shows how to lift results from the first-order fragment to the higher-order case. The second application studies two notions of stream: onetime streams (ala Common Lisp streams and Scheme ports) and reusable streams (ala Landin, Scheme). In either case a stream is characterized by the (possibly infinite) sequence it generates. Techniques for transforming definitions of sequences to definitions of streams, for memoizing, and for transforming between onetime and reusable streams preserving the underlying sequence are presented. This illustrates many aspects of reasoning about objects with memory, specification of objects with memory, and use of objects with memory as optimized versions of pure pfns.

Acknowledgements.

The first author would like to thank Furio Honsell for numerous helpful discussions. This research was partially supported by DARPA contract N00039-84-C-0211.

6. References

Felleisen, M.

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

Landin, P. J.

[1964] The mechanical evaluation of expressions, *Computer Journal*, **6**, pp. 308–320.

Mason, I. A.

[1986] The semantics of destructive Lisp, Ph.D. Thesis, Stanford University.

Mason, I. A. and Talcott, C. L.

[1989a] Equivalence of programs with function abstractions and memories, Submitted to *Lisp and Symbolic Computation.*

[1989b] Axiomatizing Operational Equivalence in the presence of Side Effects, in: *Symposium on logic in computer science*, (IEEE), (to appear).

[1989c] A sound and complete axiomatization of operational equivalence between programs with memory, Department of Computer Science, Stanford University, Technical report STAN-CS-89-1250.

[1989d] Programming, transforming and proving with function abstractions and memories, Department of Computer Science, Stanford University, Technical report STAN-CS-89-????.

Morris, J. H.

[1968] Lambda calculus models of programming languages, Ph.D. thesis, MIT.

Mosses, P.

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

Plotkin, G.

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

Reynolds, J. C.

[1972] Definitional interpreters for higher-order programming languages, in: *Proceedings, ACM national convention*, pp. 717–740.

Talcott, C.

[1985] The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation, Ph.D. Thesis, Stanford University.

[1987] Programming and proving with function and control abstractions, (Course notes)