Equivalence in functional languages with effects

IAN MASON AND CAROLYN TALCOTT
Department of Computer Science, Stanford University, Stanford, CA 94305-2095, USA

Abstract

Traditionally the view has been that direct expression of control and store mechanisms and clear mathematical semantics are incompatible requirements. This paper shows that adding objects with memory to the call-by-value lambda calculus results in a language with a rich equational theory, satisfying many of the usual laws. Combined with other recent work, this provides evidence that expressive, mathematically clean programming languages are indeed possible.

Capsule review

One of the key attractions of functional languages is that their 'clean' semantics offer the opportunity for equational reasoning about program behaviour. A large body of work under the heading of program transformation has exploited this feature to produce functional programs which are often nearly as efficient as their imperative counterparts. However, the lack of globally updateable state makes it difficult to close the efficiency gap and there is a widely held belief that the introduction of such a notion destroys equational reasoning (because of the loss of referential transparency). This paper is an important contribution to this discussion because it shows how Lisp-like destructive operations can be introduced to a language while preserving many useful equational properties.

The method employed is to establish a notion of operational approximation and associated equivalence. A number of equivalent formulations are presented, the main one being a weak form of extensionality which is introduced in section 3.1. This notion provides justification for a number of equational laws and a simulation induction principle. The ideas are well-illustrated by examples throughout the text.

1 Overview

Real programs have effects – creating new structures, examining and modifying existing structures, altering flow of control, etc. Such facilities are important not only for optimization, but also for communication, clarity, and simplicity in programming. Thus it is important to be able to reason both informally and formally about programs with effects, and not to sweep effects either to the side or under the store parameter rug.

Recent work of Talcott, Mason, Felleisen and Moggi establishes a mathematical

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

foundation 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 approximation is the pre-ordering induced by an operational semantics. Operational equivalence is the equivalence naturally associated with this 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). Operational approximation and equivalence are congruence relations on expressions, and hence closed under substitution and abstraction. Mason (1986, 1988) and Talcott (1985, 1989) 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. These calculi are simplified and extended by Felleisen and Hieb (1989). Talcott, Mason and Felleisen all apply their theories to expressing and proving properties of program constructs, and of particular programs. Moggi (1989, 1990) introduces the notion of computational monad as a framework for axiomatizing features of programming languages. Computational monads are categories with certain additional structure that accommodate a wide variety of language features including assignment, exceptions and control abstractions. An extension of the lambda-v calculus called the lambda-c calculus is presented, and shown to be valid in all computational monads.

Reduction calculi and operational equivalence 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. However, simple reduction calculi are not adequate to prove many basic equivalences in languages with effects. For example, Felleisen found it is necessary to extend his reduction calculus by meta principles (cf. the safety rule (Felleisen, 1987, Theorem 5.27, p 149). Operational equivalence 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. A key problem in developing reduction 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 the Church–Rosser property. 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 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 cannot 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 and for accessing and 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, since variables that can be assigned cannot simply be replaced by values. For example the program $(\lambda x.\mathsf{seq}(\mathsf{setq}(x,1),x))\,2$ 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 to, via setq. The presence of setq makes it impossible to substitute for variables. 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 (Mosses, 1984). 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 (1987) 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.

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, etc.), memories with cells that contain only a single atom or cell are not adequate for representing general list structures. In the higher-order case we could equally well work with simple unary cell memories. We will work with S-expression memories (memories with binary cells), as this is the natural extension of our work on the first-order case. 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.

In section 2 we define the syntax and semantics of our language. Computation is represented as a simple term rewriting system. In section 3 we give three equivalent definitions of operational approximation and equivalence. Two of the definitions are simple variants of the standard definition à la Plotkin (1975). The third definition is a weak form of extensionality, and is the key tool for proving approximation and equivalence. In particular, a simple semantic property is shown to imply operational equivalence using this form of extensionality. This property is a generalization of the notion of strong isomorphism defined for the first-order fragment in Mason (1986). As a consequence, the laws of strong isomorphism are valid for operational equivalence. For example, if one expression reduces to another then the two expressions are operationally equivalent. Several other principles for establishing operational equivalence are provided. In section 4 we define a notion of recursion operator and give two examples. 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. Using the weak form of extensionality we establish that recursion operators compute the least fixed point (with respect to operational approximation) of functionals. We also prove that recursion operators are operationally equivalent on functionals. In section 5 we use the weak form of extensionality to derive a simulation induction principle for proving equivalence of pfn objects (operations with local memory). We give two examples illustrating the application of this principle. In the first example we classify several presentations of streams by pfn objects, and prove properties of operations on such stream presentations. In the second example we define and prove several results concerning objects. Objects are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In our framework, objects are represented as pfns (closures) with mutable data bound to local variables. We show how to specify an object, how such an object behaves, and how one can represent such an object. In section 6 we relate the notions of operational equivalence and strong isomorphism in various fragments of our language. In particular, we present results that essentially characterize the difference between operational equivalence and strong isomorphism in the presence of higher-order objects. In section 7 we discuss additional related work. (An abbreviated version of this paper appears elsewhere (Mason and Talcott, 1989*b*.)

We conclude this section with a summary of notational conventions. A glossary of notations can be found in the appendix. We use the usual notation for set membership and function application. Let Y, Y_0, Y_1 be sets. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $[y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. $\boldsymbol{P}_\omega(Y)$ 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 natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbb{N}$.

2 The framework

The syntax of our language is a simple extension of that of the lambda calculus to include basic constants (atoms) and primitive operations. The semantics is given by rules for reduction to canonical form. Canonical forms consist of a syntactic representation of memory together with a value.

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 ($\mathbb{L}$ $\mathbb{U}$ $\mathbb{E}$)
The set of λ-expressions, $\mathbb{L}$, the set of value expressions, $\mathbb{U}$, and the set of expressions, $\mathbb{E}$, are defined, mutually recursively, as the least sets satisfying the following. If $a \in \mathbb{A}$, $x \in \mathbb{X}$, $u \in \mathbb{U}$, $n \in \mathbb{N}$, $e_j \in \mathbb{E}$ for $j \leqslant n$, and $\delta \in \mathbb{F}_n$ then a, x, and $\lambda x.e_0$ are in $\mathbb{U}$, $\lambda x.e_0$ is in $\mathbb{L}$ while u, $\mathsf{if}(e_0, e_1, e_2)$, $\mathsf{app}(e_0, e_1)$, and $\delta(e_1, \ldots, e_n)$ are in $\mathbb{E}$. This definition is expressed more compactly by the following system of equations:

$$\mathbb{L} = \lambda\mathbb{X}.\mathbb{E}$$

$$\mathbb{U} = \mathbb{X} + \mathbb{A} + \mathbb{L}$$

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

We will use the equational form of defining domains in the remainder of this paper. We let $a, a_0, \ldots$ range over $\mathbb{A}$, $x, x_0, \ldots, y, z, \ldots$ range over $\mathbb{X}$, $u, u_0, \ldots$ range over $\mathbb{U}$, ρ, $\rho_0, \ldots$ range over $\mathbb{L}$, and $e, e_0, \ldots$ range over $\mathbb{E}$. We call elements of $\mathbb{L}$ pfns (Partial FuNctions). 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. A *closed expression* is an expression with no free variables. We let $\mathbb{E}_\varnothing$ be the set of all closed expressions. In other words, $e \in \mathbb{E}_\varnothing$ abbreviates $e \in \mathbb{E}$ and $\mathrm{FV}(e) = \varnothing$. 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'.

The operations, $\mathbb{F}$, are partitioned into algebraic operations and memory operations. By an algebraic operation we mean a function mapping $\mathbb{A}^n$ to $\mathbb{A}$ for some $n \in \mathbb{N}$. Algebraic operations are independent of memory. A memory operation acts on its arguments returning a value and possibly modifying the memory. The unary memory operations are

$$\{atom, cell, car, cdr\} \subseteq \mathbb{F}_1$$

and the binary memory operations are

$$\{eq, cons, setcar, setcdr\} \subseteq \mathbb{F}_2.$$

The remaining operations are assumed to be algebraic.

Definition (σ)
A value substitution is a finite map σ from variables to value expressions. $\sigma, \sigma_0, \ldots$ range 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 $\sigma(x)$, again taking care not to trap variables.

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

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

We let C, C' range over $\mathbb{C}$. $C[\![e]\!]$ 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.

In order to make programs easier to read we introduce some abbreviations. Multi-ary application and abstraction is obtained by currying, application is usually represented by juxtaposition rather than explicitly using app, and let is lambda-application. Sequencing is achieved via seq, $\mathsf{seq}(e_0, \ldots, e_n)$ evaluates the expressions e_i in order, returning the value of the last expression. This can be represented using let or if. We have defined seq in terms of if. cond is the usual Lisp conditional. $\langle e_0, \ldots, e_n\rangle$ abbreviates the expression constructing a list with elements described by $e_0, \ldots, e_n$. A unary cell is the analog of an ML reference. In our language we use *mk*, *get*, *set* to represent the constructor, access and update operations for unary cells. These abbreviations are summarized as follows:

$$\lambda x_1, \ldots, x_n\,.\,e \mathrel{=:} \lambda x_1 \ldots \lambda x_n\,.\,e$$
$$e_0(e_1 \ldots e_n) \mathrel{=:} \mathsf{app}(\ldots \mathsf{app}(e_0, e_1) \ldots e_n)$$
$$\mathsf{let}\{x := e_0\}\, e \mathrel{=:} \mathsf{app}(\lambda x\,.\,e, e_0)$$
$$\mathsf{seq}(e) \mathrel{=:} e$$
$$\mathsf{seq}(e_0, \ldots, e_n) \mathrel{=:} \mathsf{if}(e_0, \mathsf{seq}(e_1, \ldots, e_n), \mathsf{seq}(e_1, \ldots, e_n))$$
$$\mathsf{cond}[\,] \mathrel{=:} \mathsf{Nil}$$
$$\mathsf{cond}[e_0 \Rightarrow e_0', e_1 \Rightarrow e_1', \ldots, e_n \Rightarrow e_n'] \mathrel{=:} \mathsf{if}(e_0, e_0', \mathsf{cond}[e_1 \Rightarrow e_1', \ldots, e_n \Rightarrow e_n'])$$
$$\langle e_1, \ldots, e_n\rangle \mathrel{=:} cons(e_1, \ldots, cons(e_n, \mathsf{Nil}) \ldots)$$
$$mk = \lambda x\,.\,cons(x, \mathsf{Nil})$$
$$get = \lambda x\,.\,car(x)$$
$$set = \lambda x\,.\,y\,.\,\mathsf{seq}(setcar(x, y), \mathsf{Nil}).$$

2.2 Semantics

Elsewhere (Mason and Talcott, in preparation) we provide two operational semantics for expressions. The first, a standard operational semantics, was based on memory structures, while the second was based on a syntactic reduction to canonical form. We present the reduction semantics here.

The operational semantics of expressions is given by a reduction relation $\overset{*}{\mapsto}$ on *descriptions* (defined below). Computation is a process of stepwise reduction of a description to a canonical form. In order to define the reduction rules and canonical forms we introduce the notions of *redex*, *reduction context* and *memory context*. Redexes describe the primitive computation steps. A primitive step is either a β-reduction, branching according to whether a test value is Nil or not, or the application of a primitive operation to a sequence of value expressions.

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

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

An expression is either a value expression or decomposes uniquely into a redex placed in a reduction context. Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the left-first, call-by-value reduction strategy of Plotkin (1975).

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

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

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

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

Definition ($\mathbb{M}$)
A memory context Γ is a context of the form

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

where $z_i \neq z_j$ when $i \neq j$. We include the possibility that $n = 0$, in which case $\Gamma = \varepsilon$. We let $\mathbb{M}$ denote the set of all such contexts and let $\Gamma, \Gamma_0, \ldots$ range over $\mathbb{M}$.

Memory contexts are syntactic representations of memory. As such, we can view memory contexts as finite maps from variables to pairs of value expressions. Hence for $\Gamma \in \mathbb{M}$ as above we define the domain of Γ to be $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$, $\Gamma(z_i) = [u_i^a, u_i^d]$ for $1 \leqslant i \leqslant n$, and we abbreviate Γ by $\{z_i := [u_i^a, u_i^d] \mid 1 \leqslant i \leqslant n\}$. Two memory contexts are considered the same if they are the same when viewed as functions. We also define the updating operation on memory contexts. $\Gamma\{z := [u_a, u_d]\}$ 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 then $\Gamma_0 \cup \Gamma_1$ 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 ($\mathbb{D}$)
The set of descriptions $\mathbb{D}$ is defined to be the set $\mathbb{M} \times \mathbb{E}$. Thus a 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. $\Gamma; e, \Gamma_0; e_0, \ldots$ range over descriptions. *Value descriptions* are descriptions of the form $\Gamma; u$ and *pfn objects* are descriptions of the form $\Gamma; \rho$ where as usual, u denotes a value expressions and ρ denotes a pfn (lambda abstraction). We call the memory context of a pfn object its *local store*.

The semantics is given by a collection of relations. The action of the memory operations is given by the primitive reduction relation, $\xrightarrow{p}$, on descriptions. $\mapsto$ is the single-step reduction relation on descriptions. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$.

We begin with the action of the memory operations: *atom* is the characteristic function – using the booleans T and Nil – of the atoms, *cell* is the characteristic function of the cells. *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 and 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 containing arbitrary values can be constructed. In particular a cell can store itself as one of its components. Finally, *eq* tests whether two values are identical. There are a number of possible choices for defining *eq* in the presence of higher-order objects. The main criteria is that we do not allow *eq* to make any non-trivial distinctions involving higher-order objects. We have chosen to define *eq* to be false when either argument is a pfn. An alternative would be to define *eq* to be true when both arguments are pfns, but false if one but not both arguments is a pfn. In either case we can define a predicate that is true on pfns and false elsewhere, in this world, and hence either version can be defined from the other.

Definition ($\xrightarrow{p}$)

The primitive reduction relation is defined as follows:

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

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

$$\Gamma; R[\![eq(u_0, u_1)]\!] \xrightarrow{p} \begin{cases} \Gamma; R[\![\mathsf{T}]\!] & \text{if } u_0 = u_1 \quad \text{and} \quad u_0, u_1 \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathsf{Nil}]\!] & \text{if } u_0 \neq u_1 \quad \text{and} \quad u_0, u_1 \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathsf{Nil}]\!] & \text{if either } u_0 \text{ or } u_1 \text{ is in } \mathbb{L} \end{cases}$$

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

$$\Gamma; R[\![car(z)]\!] \xrightarrow{p} \Gamma; R[\![u_a]\!]$$

$$\Gamma; R[\![cdr(z)]\!] \xrightarrow{p} \Gamma; R[\![u_d]\!]$$

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

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

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

Note that in the *atom* and *cell* rules, if one of the arguments is a variable not in the domain of the memory context, then the primitive reduction step is not determined. This is also the case in the *car*, *cdr*, *setcar* and *setcdr* rules when z is not in the domain of Γ.

We define single-step reduction on descriptions as follows.

Definition ($\mapsto$)

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

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

(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 Γ; $R[\![\delta(u_1, \ldots, u_n)]\!] \xrightarrow{p} \Gamma'$; $R[\![u']\!]$.

Lemma (*alpha*)

If $\Gamma; e \mapsto \Gamma_i; e_i$ for $i < 2$ then $\Gamma_0[\![e_0]\!] = \Gamma_1[\![e_1]\!]$.

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

(*alpha*) expresses the fact that $\mapsto$ (and in particular that $\xrightarrow{\mathrm{p}}$) is functional modulo alpha conversion. This makes explicit the fact that arbitrary choice in cell allocation is the same phenomenon as arbitrary choice of names of bound variables. The key distinction between $\Gamma; e$ and $\Gamma[\![e]\!]$ is that renaming of variables in $\mathrm{Dom}(\Gamma)$ is allowed in $\Gamma[\![e]\!]$ but not in $\Gamma; e$. This distinction, though somewhat technical, is important for the statement and proof of a number of results.

Definition ($\downarrow \uparrow$)
A description, $\Gamma; e$ is defined (written $\downarrow \Gamma; e$) if it evaluates to a value description. A description is undefined (written $\uparrow \Gamma; e$) if it is not defined.

$$\downarrow(\Gamma; e) \Leftrightarrow (\exists \Gamma'; u')(\Gamma; e \overset{*}{\mapsto} \Gamma'; u')$$
$$\uparrow(\Gamma; e) \Leftrightarrow \neg\downarrow(\Gamma; e).$$

In order to state and prove various results we need a form of substitution for descriptions that permits trapping of variables in the domain of the memory context, but is otherwise ordinary substitution. We write such substitutions as superscripts.

Definition ($(\Gamma; e)^\sigma$)
If $\Gamma = \{z_i := [u_i^a, u_i^d] \mid i < n\}$ and $x \notin \mathrm{Dom}(\Gamma)$ then $(\Gamma; e)^{\{x := e_0\}}$ is defined to be $\Gamma^{\{x := e_0\}}; e\{x := e_0\}$ where

$$\Gamma^{\{x := e_0\}} = \{z_i := [u_i^a\{x := e_0\}, u_i^d\{x := e_0\}] \mid i < n\}.$$

For such substitutions to yield descriptions we further require that $u_i^\alpha\{x := e_0\} \in \mathbb{U}$ for $\alpha \in \{a, d\}$ and $i < n$. Similarly, for value substitutions σ such that $\mathrm{Dom}(\sigma) \cap \mathrm{Dom}(\Gamma) = \varnothing$ we define $(\Gamma; e)^\sigma = \Gamma^\sigma; e^\sigma$ where

$$\Gamma^\sigma = \{z_i := [(u_i^a)^\sigma, (u_i^d)^\sigma] \mid i < n\}.$$

The following are some simple consequences of the syntactic computation rules.

Lemma (*cr*)
(i) Memory contexts may be moved across reduction contexts:

$$R[\![\Gamma[\![e]\!]]\!] \overset{*}{\mapsto} \Gamma; R[\![e]\!]$$

if $\mathrm{FV}(R) \cap \mathrm{Dom}(\Gamma) = \varnothing$.
(ii) Computation is uniform in free variables:

$$\Gamma; e \mapsto \Gamma'; e' \Rightarrow (\Gamma; e)^\sigma \mapsto (\Gamma'; e')^\sigma$$

if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\sigma) = \varnothing$.
(iii) Untouched memory is just carried along:

$$\Gamma; e \mapsto \Gamma'; e' \Rightarrow (\Gamma_0 \cup \Gamma); e \mapsto (\Gamma_0 \cup \Gamma'); e'$$

if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_0) = \varnothing$.

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

3 Operational approximation and equivalence

In this section we define the operational approximation and equivalence relations and study their general properties. Operational approximation ($\sqsubseteq$) is a pre-ordering relation determined by the operational semantics. Operational equivalence ($\cong$) is the corresponding equivalence relation obtained by intersecting operational approximation with its inverse. Modulo operational equivalence, operational approximation is a partial ordering with respect to definedness. Operational equivalence formalizes the notion of equivalence as black-boxes. Treating programs as black boxes requires only observing what effects and values they produce, and not how they produce them. Our definition extends the extensional equivalence relations defined by Morris (1968) and Plotkin (1975) to computation over memory structures. As shown by Abramsky (1990) and Howe (1989), operational approximation is the maximum bisimulation relation for a large class of pure functional languages.

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

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall C \in \mathbb{C} \mid C[\![e_0]\!], C[\![e_1]\!] \in \mathbb{E}_{\varnothing})(\downarrow C[\![e_0]\!] \Rightarrow \downarrow C[\![e_1]\!])$$
$$e_0 \cong e_1 \Leftrightarrow e_0 \sqsubseteq e_1 \wedge e_1 \sqsubseteq e_0.$$

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 instantiating expression always returns a value. The operational approximation relation is not trivial since T and Nil are not operationally equivalent. These observations are summarized in the following lemma.

Lemma (*congruence*)

(1) $e_0 \sqsubseteq e_1$ implies $(\forall C \in \mathbb{C})(C[\![e_0]\!] \sqsubseteq C[\![e_1]\!])$.
(2) $\downarrow e$ and $e_0 \cong e_1$ does not imply $e_0\{x := e\} \cong e_1\{x := e\}$.
(3) $\neg(\mathsf{T} \cong \mathsf{Nil})$.

Proof (*congruence*)

(1) Since for any C if C' is any closing context for $C[\![e_j]\!]$ for $j < 2$ then $C'[\![C]\!]$ is a closing context for e_j for $j < 2$.
(2) As a counterexample we have $eq(x, x) \cong \mathsf{T}$ but $eq(cons(\mathsf{T}, \mathsf{T}), cons(\mathsf{T}, \mathsf{T})) \cong \mathsf{Nil}$.
(3) The context $\mathsf{if}(\varepsilon, car(\mathsf{T}), \mathsf{T})$ will distinguish T and Nil. $\square$

The reason underlying (*congruence* (2)) is that in the case of programs with effects, returning a value is not an appropriate characterization of definedness. In particular, returning a value is not the same as being operationally equivalent to a value. This is in contrast to the purely functional case, and is due to the presence of *effects*. For example $cons(x, y)$ always returns a value, but is not operationally equivalent to a value. Similarly, an expression of the form $\Gamma[\![\lambda x.e]\!]$ is in general not operationally equivalent to a value. This distinction means that care must be taken in generalizing notions such as η-conversion and fixed-point operators (see section 4).

The η rule for the pure lambda calculus has the form $e \cong \lambda x.e(x)$ if x is not free in e. In an applied 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 ρ then the restricted η rule is not valid. If we interpret *e denotes a function* as $e \cong \lambda y.e'$, then the restricted η rule is valid.

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

Proof ($\neg\eta$)
As a counterexample we have

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

In the presence of basic data and equality tests an alternate definition of operational approximation and equivalence 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 corresponds to the definition given by Plotkin. Both definitions are equivalent in our setting, since equality on basic data is computable. This is formalized by the following:

Definition ($\sqsubseteq^0$)
For closed expressions e_0, e_1, trivial approximation ($\sqsubseteq^0$) is defined by

$$e_0 \sqsubseteq^0 e_1 \Leftrightarrow (\forall \Gamma_0; u_0)(e_0 \overset{*}{\mapsto} \Gamma_0; u_0 \Rightarrow (\exists \Gamma_1; u_1)(e_1 \overset{*}{\mapsto} \Gamma_1; u_1$$
$$\wedge ((u_0 = u_1 \in \mathbb{A}) \vee (\forall j < 2)(u_j \in \mathrm{Dom}(\Gamma_j)) \vee (u_0, u_1 \in \mathbb{L})))).$$

Theorem (*alt*) $e_0 \sqsubseteq e_1 \Leftrightarrow (\forall C \in \mathbb{C})(C[\![e_0]\!], C[\![e_1]\!] \in \mathbb{E}_{\varnothing} \Rightarrow C[\![e_0]\!] \sqsubseteq^0 C[\![e_1]\!])$.

Proof (*alt*)
The if direction is trivial. For the other direction suppose for some closing context C we have $\neg(C[\![e_0]\!] \sqsubseteq^0 C[\![e_1]\!])$. Then we can find a closing context C' such that $\downarrow(C'[\![e_0]\!])$ and $\uparrow(C'[\![e_1]\!])$. If $\uparrow(C[\![e_1]\!])$ we are done, so we assume $C[\![e_j]\!] \overset{*}{\mapsto} \Gamma_j; v_j$ for $j < 2$. If $v_0 \in \mathbb{A}$ and $v_0 \neq v_1$ then take $C' = \mathsf{if}(eq(v_0, C), car(\mathsf{T}), \mathsf{T})$. If $v_0 \in \mathrm{Dom}(\Gamma_0)$ and $v_1 \notin \mathrm{Dom}(\Gamma_1)$ then take $C' = \mathsf{if}(cell(C), car(\mathsf{T}), \mathsf{T})$. Similarly for dual cases on v_1. $\square$

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

3.1 Weak extensionality

Another characterization of operational approximation and equivalence is obtained by extending the semantic characterization of the maximum approximation relation given by Talcott (1985). This characterization states that 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 approximation relation $\sqsubseteq^{ciu}$ to mean that all closed instantiations of all uses are trivially approximate. We then show that this relation is 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*)

$$e_0 \sqsubseteq^{ciu} e_1 \Leftrightarrow (\forall \Gamma, \sigma, R)(\Gamma[\![R[\![e_0^\sigma]\!]]\!], \Gamma[\![R[\![e_1^\sigma]\!]]\!] \in \mathbb{E}_\varnothing \Rightarrow (\downarrow \Gamma[\![R[\![e_0^\sigma]\!]]\!] \Rightarrow \downarrow \Gamma[\![R[\![e_1^\sigma]\!]]\!])).$$

In this definition, Γ, σ represent the closed instantiation while R corresponds to the use.

Theorem (*ciu*) $\qquad e_0 \sqsubseteq e_1 \Leftrightarrow e_0 \sqsubseteq^{ciu} e_1.$

A sketch of the proof of (*ciu*) appears at the conclusion of this section. A direct corollary of the (*ciu*) characterization of operational approximation is the following weak form of extensionality.

Corollary (*wk.ext*)

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall \Gamma, \sigma, R)(\Gamma[\![R[\![e_0^\sigma]\!]]\!], \Gamma[\![R[\![e_1^\sigma]\!]]\!] \in \mathbb{E}_\varnothing \Rightarrow \Gamma[\![R[\![e_0^\sigma]\!]]\!] \sqsubseteq \Gamma[\![R[\![e_1^\sigma]\!]]\!]).$$

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 is a form of extensionality. When objects with memory are introduced all closed instantiations of two expressions can be equivalent, but still make essentially different use of the memory supplied so that a general context can distinguish them by supplying a memory and later modifying that memory. The notion of all closed instantiations being approximate, as well as the result just mentioned, is made explicit in the following.

Definition ($\sqsubseteq^{ci}$)

$$e_0 \sqsubseteq^{ci} e_\iota \Leftrightarrow (\forall \Gamma, \sigma)(\Gamma[\![e_0^\sigma]\!], \Gamma[\![e_1^\sigma]\!] \in \mathbb{E}_\varnothing \Rightarrow \Gamma[\![e_0^\sigma]\!] \sqsubseteq \Gamma[\![e_1^\sigma]\!]).$$

Lemma (*non.ext*) $\qquad e_0 \sqsubseteq^{ci} e_\iota$ does not imply $e_0 \sqsubseteq e_1$.

This lemma will be proved after sufficient tools have been developed.

Another 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 $e_0, e_1 \in \mathbb{E}_\varnothing$ then

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall R)(\mathrm{FV}(R) = \varnothing \Rightarrow (\downarrow R[\![e_0]\!] \Rightarrow \downarrow R[\![e_1]\!])).$$

Proof (*op.closed*)
The onlyif direction follows from (*congruence*). For the if direction, assume $\downarrow(R[\![e_0]\!]) \Rightarrow \downarrow(R[\![e_1]\!])$ for all closed R. Let Γ, R, σ be such that $\Gamma[\![R[\![e_j^\sigma]\!]]\!] \in \mathbb{E}_\varnothing$ for $j < 2$. Since e_0 and e_1 are closed, we may take $\sigma = \varnothing$. By the computation rules $\Gamma; R[\![e_j]\!]$ and $\mathsf{let}\{x := e_j\}\,\Gamma[\![R[\![x]\!]]\!]$ are equi-defined for $j < 2$ and $x \notin \mathrm{Dom}(\Gamma)$. And by assumption $\downarrow(\mathsf{let}\{x := e_0\}\,\Gamma[\![R[\![x]\!]]\!])$ implies $\downarrow(\mathsf{let}\{x := e_1\}\,\Gamma[\![R[\![x]\!]]\!])$. □

Proof (*ciu*)
The ($\Leftarrow$) direction is a trivial consequence of (*congruence*) and the fact that for any value substitution σ we can find a corresponding binding context S_σ such that $S_\sigma[\![e_j]\!] \overset{*}{\mapsto} s_j^\sigma$.

For the ($\Rightarrow$) direction assume $e_0 \sqsubseteq^{ciu} e_1$. We want to show that for any closing context C, if $C[\![e_0]\!]$ is defined, then $C[\![e_1]\!]$ is defined. To do this we introduce a notion of generalized expression, and extend this generalization to the other syntactic entities. A generalized expression $\hat{C}$ is like a context except that the holes may be decorated by generalized value substitutions. Each occurrence of a hole may have a different decoration. A generalized value expression $\hat{u}$ is a variable, an atom, or a generalized abstraction $\lambda x.\hat{C}$. A generalized value substitution $\hat{\sigma}$ maps variables to generalized values. A generalized memory context $\hat{\Gamma}$ has the form of an ordinary memory context where the values assigned to cells are generalized values. Generalized reduction contexts $\hat{R}$ are generated like ordinary reduction contexts from generalized values and expressions. To keep things straight we introduce a new hole symbol ν for the unique hole of a generalized reduction context, and we write $\hat{R}[\![e]\!]_\nu$ for the replacement of the distinguished hole ν. Generalized redexes are defined analogously to redexes, replacing expressions and values by their generalized counterparts. A generalized expression $\hat{C}$ is either a generalized value or it decomposes into a generalized reduction context and either a generalized redex, or a decorated hole $\varepsilon^{\hat{\sigma}}$. Replacement of holes in generalized expressions is defined as for ordinary contexts. The only difference is for decorated holes, where we have $(\varepsilon^{\hat{\sigma}})[\![e]\!] = e^{\hat{\sigma}[e]}$ and $\hat{\sigma}[\![e]\!](x) = (\hat{\sigma}(x))[\![e]\!]$.

We will show by computation induction that for any closing $\hat{\Gamma}; \hat{C}$, if $\hat{\Gamma}; \hat{C}[\![e_0]\!]$ is defined then $\hat{\Gamma}; \hat{C}[\![e_1]\!]$ is defined. Suppose not and choose a counterexample $\hat{\Gamma}; \hat{C}$ such that the computation of $(\hat{\Gamma}; \hat{C})[\![e_0]\!]$ has minimal length. We claim that $\hat{C}$ must decompose into $\hat{R}$ and $\varepsilon^{\hat{\sigma}}$. Otherwise if $\hat{C}$ is a generalized value we contradict undefinedness of $\hat{\Gamma}; \hat{C}[\![e_1]\!]$, and if $\hat{C}$ decomposes as $\hat{R}$ and $\hat{C}_p$ then $\hat{\Gamma}; \hat{C}$ reduces uniformly to a smaller counterexample. So assume $\hat{C} = \hat{R}[\![\varepsilon^\sigma]\!]_\nu$ and let $\hat{C}_j = \hat{R}[\![e_j^\sigma]\!]_\nu$ for $j < 2$. Then we have $(\hat{\Gamma}; \hat{C})[\![e_j]\!] = (\hat{\Gamma}; \hat{C}_j)[\![e_j]\!]$ for $j < 2$ and by the (*ciu*) hypothesis if $(\hat{\Gamma}; \hat{C}_0)[\![e_1]\!]$ is defined then $(\hat{\Gamma}; \hat{C}_1)[\![e_1]\!]$ is defined. Thus it suffices to show that $\downarrow(\hat{\Gamma}; \hat{C}_0)[\![e_0]\!] \Rightarrow \downarrow(\hat{\Gamma}; \hat{C}_0)[\![e_1]\!]$. If e_0 is not a value expression then $\hat{\Gamma}; \hat{C}_0$ steps uniformly to a smaller computation contradicting minimality. Thus we may assume e_0 is a value expression and $\hat{R}$ is not empty. Hence $\hat{R}$ has one of the following forms: $\hat{R}_0[\![\mathsf{if}(\nu, \hat{C}_a, \hat{C}_b)]\!]_\nu$, $\hat{R}_0[\![\mathsf{app}(\hat{u}_a, \nu)]\!]_\nu$, $\hat{R}_0[\![\mathsf{app}(\nu, \hat{C}_a)]\!]_\nu$, or $\hat{R}_0[\![\delta(\hat{u}^*, \nu, \hat{C}^*)]\!]_\nu$, where $\hat{u}^*$ (resp. $\hat{C}^*$) are possibly empty sequences of generalized values (resp. generalized expressions). In all

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

but the last two cases $\hat{\Gamma}; \hat{C}_0$ reduces uniformly to a smaller counterexample. In the last two cases we have a counterexample of the same computation size but with a smaller generalized context. Thus the elimination process must terminate in a contradiction to the minimality. □

3.2 Strong isomorphism

Mason (1986, 1988) defined the notion of strong isomorphism 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 or both are defined and evaluate to objects that are equal modulo the production of garbage. By garbage we mean cells constructed in the process of evaluation that are not accessible from either the result or the domain of the initial memory. A consequence of (*ciu*) is that strong isomorphism implies operational equivalence. Many useful laws of operational equivalence are in fact laws of strong isomorphism, and reasoning about strong isomorphism is often much easier than reasoning about operational equivalence.

Definition ($\simeq$)
Two expressions are strongly isomorphic, written $e_0 \simeq e_1$, if for every closing Γ, σ either both diverge or both evaluate to the same object up to production of garbage. More precisely $e_0 \simeq e_1$ just if for each Γ, σ such that $\Gamma[\![e_j^\sigma]\!] \in \mathbb{E}_\varnothing$ for $j < 2$, one of the following holds:

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

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

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

Proof (*striso*)
Assume $e_0 \simeq e_1$. By (*ciu*) we need only show $\Gamma; R[\![e_0^\sigma]\!]$ and $\Gamma; R[\![e_1^\sigma]\!]$ are equi-defined for closing Γ, R, σ. If $\Gamma; R[\![e_0^\sigma]\!]$ is defined then we can find $\Gamma_j; u$ for $j < 2$ such that $\Gamma; R[\![e_j^\sigma]\!] \overset{*}{\mapsto} \Gamma_j; R[\![u]\!]$ where Γ_j are the same modulo garbage relative to $\mathrm{Dom}(\Gamma)$ and u. Thus by (*cr*) $\Gamma_j; R[\![u]\!]$ are equi-valued modulo garbage, and hence equi-defined. □

The converse of (*striso*) is false. In particular, 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, as will be demonstrated in section 6.

An immediate corollary of (*striso*) is that operational equivalence satisfies the evaluation criteria.

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

Proof (*eval*) $\quad \Gamma; e \mapsto \Gamma'; e'$ implies $\Gamma[\![e]\!] \simeq \Gamma'[\![e']\!]$ by (*cr*). □

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

Another important property relating strong isomorphism to evaluation is the following lemma. It is an important tool for reasoning about pfn objects (Mason and Talcott, to appear (a)).

Lemma (*striso.eval*)
If $\mathrm{Dom}(\Gamma) = \bar{z}$ and $\Gamma[\![\langle \bar{z}, e_0 \rangle]\!] \simeq \Gamma[\![\langle \bar{z}, e_1 \rangle]\!]$ then for any closing $\Gamma'; \sigma$

$$(\Gamma' \cup \Gamma; e_0)^\sigma \simeq (\Gamma' \cup \Gamma; e_1)^\sigma.$$

A simple application of (*eval*) is the following lemma.

Lemma (*set.absorption*)
If z and w are distinct variables, then

(a) $\mathsf{let}\{z := cons(x, y)\}\, \mathsf{seq}(setcar(z, w), e) \simeq \mathsf{let}\{z := cons(w, y)\}\, e$
(b) $\mathsf{let}\{z := cons(x, y)\}\, \mathsf{seq}(setcdr(z, w), e) \simeq \mathsf{let}\{z := cons(x, w)\}\, e.$

To see this, note that in both cases the two sides reduce to the same description.

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, 1989*a*, *c*, to appear (*a*)). Laws (i)–(iii) correspond to the let-rules of the lambda-c calculus (Moggi, 1989).

Corollary (*laws*)

(i) $e\{x := u\} \simeq \mathsf{let}\{x := u\}\, e$
(ii) $e \simeq \mathsf{let}\{x := e\}\, x$
(iii) $R[\![\mathsf{let}\{x := e_0\}\, e_1]\!] \simeq \mathsf{let}\{x := e_0\}\, R[\![e_1]\!]$ for x not free in R
(iv) $R[\![\mathsf{if}(e_0, e_1, e_2)]\!] \simeq \mathsf{if}(e_0, R[\![e_1]\!], R[\![e_2]\!])$
(v) $\mathsf{if}(e_0, e_1, e_1) \simeq \mathsf{let}\{x := e_0\}\, e_1 \quad x \notin \mathrm{FV}(e_1)$
(vi) $\mathsf{let}\{x_0 := cons(u_0, u_1)\}\, \mathsf{let}\{x_1 := e_0\}\, e \simeq \mathsf{let}\{x_1 := e_0\}\, \mathsf{let}\{x_0 := cons(u_0, u_1)\}\, e$
if x_0 not free in e_0 and x_1 not free in u_0, u_1
(vii) $\mathsf{seq}(setcar(x, y_0), setcar(x, y_1)) \simeq setcar(x, y_1)$
$\mathsf{seq}(setcdr(x, y_0), setcdr(x, y_1)) \simeq setcdr(x, y_1)$
(viii) $\mathsf{seq}(setcar(x, y), x) \simeq setcar(x, y)$
$\mathsf{seq}(setcdr(x, y), x) \simeq setcdr(x, y)$
(ix) $\mathsf{seq}(setcdr(x_0, x_1), setcar(x_2, x_3), e) \simeq \mathsf{seq}(setcar(x_2, x_3), setcdr(x_0, x_1), e)$
(x) $setcar(cons(z, y), x) \simeq cons(x, y) \simeq setcdr(cons(x, z), y).$

Proof (*laws*)
In each case, for every closing Γ, σ, we have that $\Gamma; e^\sigma_{\mathrm{lhs}}$ and $\Gamma; e^\sigma_{\mathrm{rhs}}$ are either both undefined or reduce to a common description and hence $e_{\mathrm{lhs}} \simeq e_{\mathrm{rhs}}$. □

To illustrate the utility of these laws we prove that one can delay setting the *cdr* of a newly created cell until the cell is referenced. This is a key property used in many optimizations of list processing algorithms. Many more examples can be found in Mason and Talcott (1990, to appear *a*, *b*, *c*).

Lemma (delaying assignment)
If $w \notin \mathrm{FV}(e) \cup \{z\}$ and $\Gamma = \mathsf{let}\{w := cons(x,y)\}[\![\,]\!]$ then

$$\Gamma[\![\mathsf{seq}(setcdr(w,z),e,e')]\!] \simeq \Gamma[\![\mathsf{seq}(e,setcdr(w,z),e')]\!].$$

Proof (delaying assignment)

$$\begin{array}{lll} \mathsf{let}\{w := cons(x,y)\}\,\mathsf{seq}(setcdr(w,z),e,e') & & \\ \quad \simeq \mathsf{let}\{w := cons(x,z)\}\,\mathsf{seq}(e,e') & \text{by } (set.absorption) & \\ \quad \simeq \mathsf{seq}(e,\mathsf{let}\{w := cons(x,z)\}\,e') & \text{by } (laws.v,vi) & \\ \quad \simeq \mathsf{seq}(e,\mathsf{let}\{w := cons(x,y)\}\,\mathsf{seq}(setcdr(w,z),e')) & \text{by } (set.absorption) & \\ \quad \simeq \mathsf{let}\{w := cons(x,y)\}\,\mathsf{seq}(e,setcdr(w,z),e') & \text{by } (laws.v,vi). & \square \end{array}$$

Corollary (gc)
If Γ is memory context such that $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(e) = \varnothing$ then $\Gamma[\![e]\!] \simeq e$. In other words garbage can be collected.

Proof (gc)
If $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(e) = \varnothing$ then by (cr) $\Gamma[\![e]\!] \simeq e$. $\square$

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

Proof (η)
Assume $e \cong \lambda x.e'$ then

$$\begin{array}{rll} \lambda x.e(x) & \cong \lambda x.(\lambda x.e') & \text{by } (congruence) \text{ and lemma hypothesis} \\ & \cong \lambda x.e' & \text{by } (laws.i) \\ & \cong e & \text{by lemma hypothesis.} \quad \square \end{array}$$

Corollary (ξ)
If $e_j \cong \lambda x.e'_j$ for $j < 2$ and $\mathsf{app}(e_0,x) \cong \mathsf{app}(e_1,x)$ then $e_0 \cong e_1$.

Proof (ξ)
Assume $e_j \cong \lambda x.e'_j$ for $j < 2$ and $\mathsf{app}(e_0,x) \cong \mathsf{app}(e_1,x)$. Then

$$\begin{array}{rll} e_0 & \cong \lambda x.e_0(x) & \text{by } (\eta) \\ & \cong \lambda x.e_1(x) & \text{by } (congruence) \text{ and hypothesis} \\ & \cong e_1 & \text{by } (\eta). \quad \square \end{array}$$

3.3 Using weak extensionality

To see how (ciu) can be used we outline three methods for proving approximation. The first method deals with the special case of proving approximation of pfns, the second method deals with proving approximation of pfn objects – pfns with local

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

memory, and the third deals with the general case. Each of the methods amounts to finding a strengthening of the statement of (*ciu*) so that computation induction will work.

Lemma (*ciu.i*)
For $\rho_0, \rho_1 \in \mathbb{L}$, a method for proving $\rho_0 \sqsubseteq \rho_1$ is to show by computation induction that if $(\Gamma; e)^{\{p=\rho_0^\sigma\}}$ is defined then $(\Gamma; e)^{\{p=\rho_1^\sigma\}}$ is defined for all Γ, e, p, σ such that (i) $p \notin \mathrm{Dom}(\Gamma)$, (ii) $\mathrm{FV}(\Gamma[\![e]\!]) \subseteq \{p\}$ and (iii) $\mathrm{FV}(\rho_j^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$. We call (i)–(iii) the (*ciu.i*) conditions for ρ_0, ρ_1.

Proof (*ciu.i*)
For any Γ, R, σ closing ρ_0, ρ_1 we have $\Gamma; R[\![\rho_j^\sigma]\!] = (\Gamma; R[\![p]\!])^{\{p=\rho_j^\sigma\}}$ for any $p \notin \mathrm{Dom}(\Gamma)$. Thus by definition of $\sqsubseteq^{ciu}$ and (*ciu*) we are done. □

Lemma (*ciu.ii*)
For $\rho_0, \rho_1 \in \mathbb{L}$, a method for proving $\Gamma_0[\![\rho_0]\!] \sqsubseteq \Gamma_1[\![\rho_1]\!]$ is to show by computation induction that if $(\Gamma \cup \Gamma_0^\sigma; e)^{\{p=\rho_0^\sigma\}}$ is defined then $(\Gamma \cup \Gamma_1^\sigma; e)^{\{p=\rho_1^\sigma\}}$ is defined for all Γ, e, p, σ such that: (i) $p \notin \mathrm{Dom}(\Gamma) \cup \mathrm{Dom}(\Gamma_0) \cup \mathrm{Dom}(\Gamma_1)$, (ii) $\mathrm{Dom}(\Gamma) \cap \mathrm{Dom}(\Gamma_j) = \varnothing$ for $j < 2$, (iii) $\mathrm{FV}(\Gamma[\![e]\!]) \subseteq \{p\}$, and (iv) $\mathrm{FV}(\Gamma_j[\![\rho_j]\!]^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$. We call (i)—(iv) the (*ciu.ii*) conditions for $\Gamma_1; \rho_0$, $\Gamma_1; \rho_1$.

Proof (*ciu.ii*)
Note that the (*ciu.ii*) conditions imply that variables in the range of σ may be trapped by Γ but not by Γ_j. Furthermore, if $(\mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R)) \cap \mathrm{Dom}(\Gamma_j) = \varnothing$ for $j < 2$, then

$$\Gamma; R[\![(\Gamma_j[\![p_j]\!])^\sigma]\!] \overset{*}{\mapsto} \Gamma \cup \Gamma_j^\sigma; R[\![p_j^\sigma]\!].$$

Thus by (*ciu*) we need only show that if $\Gamma \cup \Gamma_0^\sigma; R[\![p_0^\sigma]\!]$ is defined then $\Gamma \cup \Gamma_1^\sigma; R[\![p_1^\sigma]\!]$ is defined for all closing Γ, R, σ such that for $j < 2$, $\mathrm{Dom}(\Gamma) \cap \mathrm{Dom}(\Gamma_j) = \varnothing$. □

Note that (*ciu.i*) is a special case of (*ciu.ii*) with $\mathrm{Dom}(\Gamma_0) = \mathrm{Dom}(\Gamma_1) = \varnothing$. As an example of the application of (*ciu.ii*) we prove (*non.ext*).

Lemma (*non,ext*) $\quad e_0 \sqsubseteq^{ct} e_1$ does not imply $e_0 \sqsubseteq e_1$.

Proof (*non.ext*)
A counterexample is

$$\begin{aligned} e_0 &= \mathsf{seq}(setcar(c, \mathrm{I}), \mathrm{I}) \\ e_1 &= \mathsf{seq}(setcar(c, \mathrm{I}), \lambda x\,.\,car(c)(x)) \\ \mathrm{I} &= \lambda x\,.\,x. \end{aligned}$$

To see this, note that for any closing Γ, σ we have either $c \notin \mathrm{Dom}(\Gamma)$ and $\Gamma; e_j^\sigma$ is undefined for $j < 2$, or $\Gamma; e_0^\sigma \simeq \mathrm{I}$ and $\Gamma; e_1^\sigma \simeq \{c := [\mathsf{I}, \mathsf{Nil}]\}\lambda x\,.\,car(c)(x)$ by (*eval*) and (*gc*). Using (*ciu.ii*) we have $\lambda x\,.\,x \cong \{c := [\mathsf{I}, \mathsf{Nil}]\}\lambda x\,.\,car(c)(x)$ and hence $e_0 \sqsubseteq^{ct} e_1$. On the other hand, for

$$C = \mathsf{let}\{c := cons(\mathsf{Nil}, \mathsf{Nil})\}\,\mathsf{let}\{p := \varepsilon\}\,\mathsf{seq}(setcar(c, \mathsf{Nil}), p(\mathsf{Nil}))$$

we have $C[\![e_0]\!]$ is defined and $C[\![e_1]\!]$ is undefined so $e_0 \not\sqsubseteq e_1$. □

Lemma (*ciu.iii*)
A method for proving $e_0 \sqsubseteq e_1$ is to show by computation induction that if $(\Gamma; e)^{\{x=e_0^\sigma\}}$ is defined then $(\Gamma; e)^{\{x=e_1^\sigma\}}$ is defined for all Γ, e, x, σ such that (i) $x \notin \mathrm{Dom}(\Gamma)$, (ii) FV $(\Gamma[\![e]\!]) \subseteq \{x\}$, (iii) $\mathrm{FV}(e_j^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$, and (iv) if $\Gamma(z) = [u_a, u_d]$ then neither u_a nor u_d is x – i.e. x occurs free in the range of Γ only inside lambda abstractions. We call (i)–(iv) the (*ciu.iii*) conditions for e_0, e_1. Condition (iv) insures that $(\Gamma; e)^{\{x=e_j^\sigma\}}$ is a description.

Proof (*ciu.iii*)
As for (*ciu.i*). □

4 Recursion pfns

The notion of *recursion operator* was introduced by Talcott (1985). A recursion operator computes the least fixed point (with respect to operational approximation) of functionals, and thus provides a mechanism for definition by recursion. The definition of recursion operator identifies the essential properties needed to prove the least-fixed-point property and captures the essence of minimality in computational terms, namely that recursive calls are sub-computations. To define a notion of recursion operator, one must first determine the class of objects of which one can meaningfully compute fixed points. In the pure call-by-value world these are clearly those objects that describe maps from functions to functions, i.e. expressions of the form (modulo operational equivalence) $\lambda f.\lambda x.e$. For any recursion operator *rec*, the fixed-point property implies that $rec(\lambda f, x.e)(x) \cong e\{f{:=}\, rec(\lambda f, x.e)\}$.

In order to extend the notion of recursion operator to the world of memories we need to determine the analog of functional (i.e. meaningful arguments for a fixed point operation). Recall that in the presence of memory effects, there is a distinction between expressions that are equivalent to a value, and expressions that always return a value, since the latter may have observable effects. Thus there are two possibilities for meaningful objects to compute fixed points of: (i) functionals as in the non-memory case, i.e. value expressions of the form $\lambda f, x.e$ or (ii) objects of the form $\Gamma[\![\lambda f, x.e]\!]$. To see that (ii) is not a reasonable choice, let *rec* be a candidate for a recursion operator. In particular, the fixed-point equation given above must hold. Also let *nconc* be the usual destructive list appending operation. If

$$\varphi = \lambda f, x.\,\mathsf{if}(eq(x, c), \mathsf{T}, \mathsf{seq}(nconc(a, cons(\mathsf{Nil}, \mathsf{Nil})), f(c)))$$

and $\Gamma = \{c{:=}\,[\mathsf{Nil}, \mathsf{Nil}]\}$, then by computation

$$rec(\Gamma[\![\varphi]\!])(\mathsf{Nil}) \cong \mathsf{seq}(nconc(a, cons(\mathsf{Nil}, \mathsf{Nil})), \mathsf{T})$$

$$\Gamma[\![\varphi]\!](rec(\Gamma[\![\varphi]\!])) \cong \mathsf{seq}(nconc(a, cons(\mathsf{Nil}, \mathsf{Nil})), nconc(a, cons(\mathsf{Nil}, \mathsf{Nil})), \mathsf{T}).$$

These two expressions can be distinguished by a context that binds a to a cell with contents $[\mathsf{Nil}, \mathsf{Nil}]$ and produces distinguishable values depending on the length of a. Thus we take option (i).

Although the functionals of which we compute fixed points have no local memory, a recursion operator may create local store, and hence fixed points themselves will be

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

pfn objects (pfns with local store). In addition, a functional may have free variables that refer to store created prior to the fixed-point computation, and thus not recreated on each recursive cell.

Definition (*recnop*)
A closed lambda expression *rec* is a recursion operator if there exist Γ, $\rho \in \mathbb{L}$, $p \notin \mathrm{Dom}(\Gamma)$, such that $\Gamma;\rho$ is a pfn object with distinguished variable p (i.e. $\mathrm{FV}(\Gamma[\![\rho]\!]) \subseteq \{p\}$) and the following two conditions hold:

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

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

We call $\Gamma;\rho$ the associated fixed-point template for *rec* (with parameter p) and we use the notation $\Gamma_\varphi;\rho_\varphi$ for $(\Gamma;\rho)^{\{p=\varphi\}}$ always assuming that $\Gamma;\rho$ has been chosen so that $\mathrm{Dom}(\Gamma) \cap (\mathrm{FV}(\varphi) \cup \{p\}) = \varnothing$. 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. The precise form of (ii) was chosen to simplify the presentation and the proof of the least-fixed-point property. Many operators will be equivalent to a recursion operator without satisfying (ii) as formulated. What is essential is that there is a smaller computation of a suitable form.

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

The proof of (*recn*) is given at the end of this section. First we discuss some consequences and give two examples of recursion operators.

As a consequence of the recursion theorem functional equations can be solved using (any) recursion operator. We write $f(x_1, \ldots, x_n) \leftarrow e$ for $f = rec(\lambda f.\lambda x_1, \ldots, \lambda x_n.e)$. It is straightforward, but tedious, to extend this to mutually recursively defined functions and we use similar notation to express least solutions to systems of equations.

A corollary of the recursion theorem is that parameters can be moved across the recursion operator (cf. Talcott, 1985, IV.4.2, VI.2.2).

Corollary (*param.rec*)
If *rec* is a recursion operator then

$$\lambda z.rec(\lambda f, x.F(z, f, x)) \cong rec(\lambda g, z, x.F(z, g(z), x)).$$

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

4.4 Example recursion pfns

Two examples of recursion operators are rec_v – a conventional call-by-value fixed-point combinator, and rec_m – a recursion operator in the spirit of letrec (Landin, 1964), and the Scheme labels construct (Steele and Sussman, 1975). rec_v uses self-application to create the recursive self-reference rec_m uses the ability to create and update cells to create the necessary self-reference.

Definition (*recv*)
The recursion combinator rec_v is defined by

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

Lemma (*recv*)
rec_v is a recursion operator.

Proof (*recv*)
The fixed-point template for rec_v is

$$\varnothing\,; \lambda x.p(\pi_p(\pi_p), x)$$

where $\pi_p = \lambda h.\lambda x.p(h(h), x)$. □

An alternative method for representing recursive definitions is by constructing a *self-referential* loop using destructive memory operations. The method is essentially identical to the one suggested by Landin (1964). It is similar to the Scheme labels construct. It also corresponds in a strong sense to the Lisp implementation of recursion using defun – i.e. to having a separate environment for function symbols where expressions in the defining bodies can refer to this environment (McCarthy *et al.*, 1962).

Definition (*rec_m*)
The memory recursion operator rec_m is defined by

$$rec_m = \lambda p.\,\mathsf{let}\{z := mk(\mathsf{Nil})\}\,\mathsf{seq}(set(z, \lambda x.p(get(z), x)), get(z)).$$

Lemma (*recm*)
rec_m is a recursion operator.

Proof (*recm*)
The fixed-point template for rec_m is

$$\{z := [\lambda x.p(get(z), x), \mathsf{Nil}]\}\,; \lambda x.p(get(z), x). \qquad \square$$

To illustrate the remark above about condition (ii) in the definition of recursion operator define

$$rec'_m = \lambda p.\,\mathsf{let}\{z := mk(\mathsf{Nil})\}\,\mathsf{seq}(set(z, p(\lambda x.get(z)(x))), \lambda x.get(z)(x)).$$

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

Then by (*eta*), $rec'_m(\varphi) \cong rec_m(\varphi)$ for any functional φ. But rec'_m does not satisfy the second recursion operator criteria in the form given.

4.2 Proof of the recursion theorem

Proof (*recn*)
Let *rec* and *rec'* be recursion operators. Let $\Gamma;\rho$ be the associated fixed-point template for *rec* (with parameter p), let φ be $\lambda f, x.e_F$ and let ψ be $\Gamma_1[\![\lambda x.e_1]\!]$. We want to show

$$\begin{aligned}
&(\text{fix}) \quad rec(\varphi) \cong \varphi(rec(\varphi))\\
&(\text{min}) \quad \varphi(\psi) \sqsubseteq \psi \Rightarrow rec(\varphi) \sqsubseteq \psi\\
&(\text{eq}) \quad rec(\varphi) \cong rec'(\varphi).
\end{aligned}$$

(*fix*) In the absence of memory we simply note that by computation $rec(\varphi)(x) \cong e_F\{f{:=}\rho_\varphi\} \cong \varphi(rec(\varphi), x)$. Then by congruence $\lambda x.rec(\varphi)(x) \cong \lambda x.\varphi(rec(\varphi), x)$ and by eta conversion we are done. However, as we noted above the eta rule does not apply to pfn objects, and so we have to work harder. By (*eval*) and the definition of recursion operator $rec(\varphi) \cong \Gamma_\varphi[\![\rho_\varphi]\!]$ and $\varphi(rec(\varphi)) \cong \Gamma_\varphi[\![\lambda x.e_F\{f{:=}\rho_\varphi\}]\!]$. We apply (*ciu.ii*) with $\Gamma_0;\psi_0 = \Gamma_\varphi;\rho_\varphi$ and $\Gamma_1;\psi_1 = \Gamma_\varphi;\lambda x.e_F\{f{:=}\rho_\varphi\}$. Note that for each σ under consideration $(\Gamma_\varphi)^\sigma = \Gamma_{\varphi^\sigma}$, $(\rho_\varphi)^\sigma = \rho_{\varphi^\sigma}$, and $(\lambda x.e_F\{f{:=}\rho_\varphi\})^\sigma = \lambda x.e_F^\sigma\{f{:=}\rho_{\varphi^\sigma}\}$. Thus we want to show $(\Gamma' \cup \Gamma_{\varphi^\sigma}; e)^{\{r=\rho_{\varphi^\sigma}\}}$ and $(\Gamma' \cup \Gamma_{\varphi^\sigma}; e)^{\{r=\lambda x.e^\sigma\{f=\rho_{\varphi^\sigma}\}\}}$ are equi-defined for Γ', e, r, σ satisfying the (*ciu.ii*) conditions. The only interesting case is when $e = \mathsf{app}(r, u)$. The others terminate or step uniformly to smaller computations assuming r ranges over $\mathbb{L}$. In this case we have

$$\Gamma' \cup \Gamma_{\varphi^\sigma}; \mathsf{app}(\rho_{\varphi^\sigma}, u) \overset{*}{\mapsto} \Gamma' \cup \Gamma_{\varphi^\sigma}; e_F^\sigma\{f{:=}\rho_{\varphi^\sigma}, x{:=}u\}$$

$$\Gamma' \cup \Gamma_{\varphi^\sigma}; \mathsf{app}(\lambda x.e_F^\sigma\{f{:=}\rho_{\varphi^\sigma}\}, u) \overset{*}{\mapsto} \Gamma' \cup \Gamma_{\varphi^\sigma}; e_F^\sigma\{f{:=}\rho_{\varphi^\sigma}, x{:=}u\}.$$

Thus reducing the problem to smaller computations. □

(*min*) Assume $\varphi(\psi) \sqsubseteq \psi$. We apply (*ciu.ii*) with $\Gamma_0;\psi_0 = \Gamma_\varphi;\rho_\varphi$ and $\Gamma_1;\psi_1 = \Gamma_1;\lambda x.e_1$. Thus we want to show that if $(\Gamma' \cup \Gamma_{\varphi^\sigma}; e)^{\{r=\rho_{\varphi^\sigma}\}}$ is defined then $(\Gamma' \cup \Gamma_1^\sigma; e)^{\{r=(\lambda x.e_1)^\sigma\}}$ is defined for Γ', ε, r, σ satisfying the (*ciu.ii*) conditions. Again, the only interesting case is when $e = \mathsf{app}(r, u)$ and we have as before $\Gamma' \cup \Gamma_{\varphi^\sigma}; \rho_{\varphi^\sigma}(\mathrm{u}) \overset{*}{\mapsto} \Gamma' \cup \Gamma_{\varphi^\sigma}; e_F^\sigma\{f{:=}\rho_{\varphi^\sigma}, x{:=}u\}$ and by the induction hypothesis if $(\Gamma' \cup \Gamma^{\varphi^\sigma}; e)^{\{r=\rho^{\varphi^\sigma}\}}$ is defined then $(\Gamma' \cup \Gamma_1^\sigma; R[\![e_F^\sigma\{x{:=}u, f{:=}r\}]\!])^{\{r=(\lambda x.e_1)^\sigma\}}$ is defined and by the assumption on ψ $(\Gamma' \cup \Gamma_1^\sigma; R[\![\mathsf{app}(r, u)]\!])^{\{r=(\lambda x.e_1)^\sigma\}}$ is defined. □

(*eq*) Similar to (*fix*). □ □

5 Simulation induction

It is often the case that the intuitive reason that two pfn objects are equivalent is that replacing one by the other results in *similar* computations. Here similar means that the computations have the same steps if one treats applications of the pfn objects

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

under consideration as single steps. In general, we need to consider families of similar pfn objects and computations that are related by replacing objects from one family by corresponding objects from the other family.

In this section we derive a principle we call *simulation induction* for proving equivalence of corresponding pairs of pfn objects. We begin by defining the notion of simulation correspondence. A simulation correspondence is a family of pairs of pfn objects that describe similar computations. We show that corresponding pfn objects in a simulation correspondence are operationally equivalent, and we derive a principle called simulation induction that can be used to prove that a family of pairs of pfn objects is a simulation correspondence.

To simplify the statement of hygiene conditions we assume that variables are partitioned into four disjoint (and infinite) collections: ${}^{c}\mathbb{X}$ for general cells, ${}^{o}\mathbb{X}$ for cells local to pfn objects of interest, ${}^{v}\mathbb{X}$ for general values, and ${}^{p}\mathbb{X}$ for pfn parameters.

Definition (*object correspondence*)
An object correspondence is a collection of pairs of pfn objects satisfying certain simple hygiene conditions. Formally object correspondences are the subsets $\mathcal{O}$ of ($\mathbb{M}$; $\mathbb{L} \sim \mathbb{M}$; $\mathbb{L}$) such that if $\Gamma_0;\rho_0 \sim \Gamma_1;\rho_1$ is a member of $\mathcal{O}$ then $\mathrm{Dom}(\Gamma_j) \subset {}^{o}\mathbb{X}$ and $\mathrm{FV}(\Gamma_j[\![\rho_j]\!]) \subset {}^{v}\mathbb{X}$ for $j < 2$. (The sign $\sim$ as used here is jsut to be thought of as a pair constructor.)

We let $\mathcal{O}$ be an object correspondence.

Definition (*local correspondence*)
An $\mathcal{O}$-local correspondence is a quadruple $(\Gamma_0;\pi_0 \sim \Gamma_1;\pi_1)$ where Γ_0, Γ_1 are memory contexts, and for some finite subset P of ${}^{p}\mathbb{X}$, π_0, π_1 are maps from P to $\mathbb{L}$ such that $\Gamma_0;\pi_0(p) \sim \Gamma_1;\pi_1(p)$ is a member of $\mathcal{O}$ for each $p \in P$.

We extend value substitutions homomorphically to local correspondence maps π writing $\sigma(\pi)$ for the result of applying σ to π. Thus $\sigma(\pi)(p) = \pi(p)^{\sigma}$ for $p \in \mathrm{Dom}(\pi)$.

Definition (*simulation correspondence*)
An object correspondence $\mathcal{O}$ is a simulation correspondence if for each Γ, e, σ, Γ_0, Γ_1, π_0, π_1, P such that

(s.i) $\mathrm{Dom}(\Gamma) \subset {}^{c}\mathbb{X}$,
(s.ii) $\mathrm{FV}(\Gamma[\![e]\!]) = P \subset {}^{p}\mathbb{X}$,
(s.iii) $(\Gamma_0;\pi_0 \sim \Gamma_1;\pi_1)$ is a $\mathcal{O}$-local correspondence with $\mathrm{Dom}(\pi_0) = P$,
(s.iv) $\mathrm{FV}(\Gamma_j[\![\pi_j(p)]\!]) \subseteq \mathrm{Dom}(\sigma) \subset {}^{v}\mathbb{X}$ for $p \in P$,
(s.v) $\mathrm{FV}(\mathrm{Rng}(\sigma)) \subset \mathrm{Dom}(\Gamma)$

either $(\Gamma \cup \Gamma_j^{\sigma}; e)^{\sigma(\pi_j)}$ is undefined for $j < 2$ or there exist $\Gamma'; u$, σ', Γ'_0, Γ'_1, π'_0, π'_1 satisfying the above conditions (and, without loss of generality, π'_j is an extension of π_j and σ' is an extension of σ) such that

$$(\Gamma \cup \Gamma_j^{\sigma}; e)^{\sigma(\pi_j)} \simeq (\Gamma' \cup \Gamma_j'^{\sigma'}; u)^{\sigma'(\pi'_j)}$$

for $j < 2$.

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

Theorem (simulation equivalence)
If $\mathcal{O}$ is a simulation correspondence then $(\Gamma_0; \rho_0 \cong \Gamma_1; \rho_1)$ for each $(\Gamma_0; \rho_0 \sim \Gamma_1; \rho_1)$ in $\mathcal{O}$.

Proof (simulation equivalence)
By (*ciu*). □

Theorem (simulation induction)
$\mathcal{O}$ is a simulation correspondence if for each Γ, u, σ, Γ_0, Γ_1, π_0, π_1, P such that (s.i)–(s.v) hold with e replaced by u then either $(\Gamma \cup \Gamma_j; p(u))^{\sigma(\pi_j)}$ is undefined for $j < 2$ or there exist $\Gamma'; u'$, σ', Γ'_0, Γ'_1, π'_0, π'_1, P' satisfying the conditions (s.i)–(s.v) with π'_j an extension of π_j and σ' and extension of σ such that

$$(\Gamma \cup \Gamma_j; p(u))^{\sigma(\pi_j)} \simeq (\Gamma' \cup \Gamma'_j; u')^{\sigma'(\pi'_j)}$$

for $j < 2$.

Proof (simulation induction)
By computation induction. Assume the condition of simulation induction statement holds. Let Γ, e, σ, Γ_0, Γ_1, π_0, π_1, P satisfy the conditions (s.i)–(s.v) of the simulation correspondence hypothesis. We will show that if $(\Gamma \cup \Gamma_0; e)^{\sigma(\pi_0)}$ is defined then $(\Gamma \cup \Gamma_1; e)^{\sigma(\pi_1)}$ is defined. The other direction is symmetric. Assume $(\Gamma \cup \Gamma_0; e)^{\sigma(\pi_0)}$ is defined. If e is a value expression we are done. If e has the form $R[\![e_r]\!]$ where e_r is a redex and not of the form $u_f(u_a)$ with u_f a variable, then both descriptions step uniformly to smaller corresponding computations. If e_r has the form $u_f(u_a)$ with u_f a variable then we are done by the simulation induction condition and (*cr*). Note that strong isomorphism to a value description implies reduction to that value description, modulo garbage collection. □

5.1 Streams

As an illustration of the application of simulation induction we consider two classes of streams and relations between them. Streams are mechanisms for generating potentially infinite sequences. We will focus on streams of pure elements – values that (up to equivalence) are independent of memory. A pure sequence is a pfn that computes a total function from $\mathbb{N}$ to pure values. We consider two kinds of stream: *onetime* and *reusable*. A onetime stream is a pfn object which when queried returns the next element of the sequence being generated and updates its local store. The nth query produces the nth stream element and that pfn object cannot in general be reused to generate the same element again. A reusable stream is a pfn object which when queried produces a pair consisting of the next stream element and a pfn representing the remainder of the stream. The behaviour of the pfn object itself is unchanged and repeated query will return the same result.

To make these notions precise we define a collection of operations: *s2o*, *s2r*, *o2r*, *r2o*, and *memo*. *s2o* maps pure sequences into onetime streams. A onetime stream is a pfn object equivalent to *s2o*(f) for some pure sequence f and f is the sequence generated by that stream. *s2r* maps pure sequences into reusable streams. A reusable

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

stream is a pfn object equivalent to $s2r(f)$ for some pure sequence f and f is the sequence generated by that stream. $o2r$ maps onetime streams into reusable streams preserving the sequence generated. $r2o$ maps reusable streams into onetime streams preserving the sequence generated. $memo$ maps reusable streams to reusable streams preserving the sequence generated and memorizing the elements computed so far, so the second request for a given element looks it up rather than recomputing it.

Definition (*stream operations*)

$$
\begin{aligned}
&s2o(f) \leftarrow s2oa(f, mk(0))\\
&s2oa(f, c) \leftarrow \lambda d.\,\mathsf{let}\{n := get(c)\}\,\mathsf{seq}(set(c, n+1), f(n))\\
&s2r(f) \leftarrow s2ra(f, 0)\\
&s2ra(f) \leftarrow \lambda d.\,cons(f(n), s2ra(f, n+1))\\
&o2r(s) \leftarrow o2ra(cons(s, \mathsf{Nil}))\\
&o2ra(c) \leftarrow \lambda d.\,\mathsf{seq}(otouch(c), cons(car(c), o2ra(cdr(c))))\\
&otouch(c) \leftarrow \mathsf{if}(cdr(c),\\
&\qquad\qquad \mathsf{Nil},\\
&\qquad\qquad \mathsf{let}\{x := \mathsf{app}(car(c), \mathsf{Nil})\}\\
&\qquad\qquad\quad \mathsf{seq}(setcdr(c, cons(car(c), \mathsf{Nil})), setcar(c, x))))\\
&r2o(s) \leftarrow r2oa(mk(s))\\
&r2oa(c) \leftarrow \lambda d.\,\mathsf{let}\{z := \mathsf{app}(get(c), \mathsf{Nil})\}\,\mathsf{let}\{x := car(z)\}\,\mathsf{let}\{s := cdr(z)\}\\
&\qquad\qquad\quad \mathsf{seq}(set(c, s), x)\\
&memo(r) \leftarrow mema(cons(r, \mathsf{Nil}))\\
&mema(c) \leftarrow \mathsf{seq}(rtouch(c), cons(car(c), mema(cdr(c))))\\
&\mathsf{rtouch(c)} \leftarrow \mathsf{if}(cdr(c),\\
&\qquad\qquad \mathsf{Nil},\\
&\qquad\qquad \mathsf{let}\{z := \mathsf{app}(car(c), \mathsf{Nil})\}\,\mathsf{let}\{x := car(z)\}\,\mathsf{let}\{s := cdr(z)\}\\
&\qquad\qquad\quad \mathsf{seq}(setcdr(c, cons(s, \mathsf{Nil})), setcar(c, x)))).
\end{aligned}
$$

Definition (*onetime and reusable streams*)
Let f be a pure sequence. A onetime stream generating f is a pfn object operationally equivalent to $s2o(f)$. A reusable stream generating f is a pfn object operationally equivalent to $s2r(f)$.

Theorem (*s.o.r*)
For f a pure sequence

(i) $o2r$ maps reusable to onetime streams preserving the sequence generated:

$$o2r(s2o(f)) \cong s2r(f).$$

(ii) $r2o$ maps onetime to reusable streams preserving the sequence generated:

$$r2o(s2r(f)) \cong s2o(f).$$

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

(iii) *memo* maps reusable to reusable streams preserving the sequence generated:

$$memo(s2r(f)) \cong s2r(f).$$

Corollary (*o.r*)
o2r and *r2o* are inverses on their intended domains.

(i) If ψ is a onetime stream then $r2o(o2r(\psi)) \cong \psi$.
(ii) If ψ is a reusable stream then $o2r(r2o(\psi)) \cong \psi$.

Proof (*s.o.r*)
(i) Let

$$\mathcal{O} = \{\Gamma_n; o2ra(y_j) \sim s2ra(f,j) \mid n \in \mathbb{N}, j \leqslant n)$$

where

$$\Gamma_n = \{y := [n, \mathsf{Nil}], y_n := [s2oa(f,y), \mathsf{Nil}], y_j := [f(j), y_{j+1}] \mid j < n\}.$$

Then by simulation induction we have $\mathcal{O}$ is a simulation correspondence. Since by (*eval*) $o2r(s2o(f)) \cong \Gamma_0; o2ra(y_0)$ and $s2r(f) \cong s2ra(f,0)$ we are done.
(ii) Similar to (i) letting

$$\mathcal{O} = \{\{y := [s2ra(f,n), \mathsf{Nil}]\}; r2oa(y) \sim \{y := [n, \mathsf{Nil}]\}; s2oa(f,y) \mid n \in \mathbb{N}, j \leqslant n\}.$$

(iii) Similar to (i) letting

$$\mathcal{O} = \{\Gamma_n; mema(y_j) \sim s2ra(f,j) \mid n \in \mathbb{N}, j \leqslant n\}$$

where

$$\Gamma_n = \{y_n := [s2ra(f,n), \mathsf{Nil}], y_j := [f(j), y_{j+1}] \mid j < n\}. \qquad \square$$

5.2 Specifications, behaviours and objects

As a further indication of how our theory can be applied, we consider a generalization of the notion of stream which we call *object*. Objects are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In our framework objects are represented as pfns (closures) with mutable data bound to local variables. In the current state of development the framework treats only sequential computation. However, the techniques such as simulation induction and constraint propagation (cf. Mason and Talcott, to appear *c*), have been designed with the goal in mind of treating objects that exist in and communicate with other objects in an open distributed system. In particular, we aim to provide a basis for both informal and formal reasoning about actors and similar systems (Agha, 1986; Hewitt, 1977; Yonezawa, 1990). We apply these methods in (Mason and Talcott, to appear *b*) to give a formal derivation of an optimized speicalized window editor from generic specifications of its components.

We specify an object by a set of local parameters, a message parameter, and a sequence of message handlers. A message handler consists of a test function, a reply function and a list of updating functions (one for each parameter). The functions take as arguments the message and current value of the local parameters. Upon receipt of a message, the first handler whose test is true is invoked. The local parameters are updated according to the update expressions and the reply is computed by the reply function. (Evaluation of the test, updating, and reply functions should have no

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

(visible) effect.) A specification S with k local parameters $\bar{x}$, message parameter *msg*, and ith message handler with test function t_i, reply function r_i, and an updating function $u_{i,j}$ for $1 \leqslant j \leqslant k$ is written in the form

Definition (S)

$$
\begin{aligned}
S = {} & (\bar{x})(msg) \\
& [t_0(\bar{x}, msg) \Rightarrow r_0(\bar{x}, msg), u_{0,1}(\bar{x}, msg), \ldots, u_{0,k}(\bar{x}, msg) \\
& \ldots \\
& t_m(\bar{x}, msg) \Rightarrow r_m(\bar{x}, msg), u_{m,1}(\bar{x}, msg), \ldots, u_{m,k}(\bar{x}, msg)].
\end{aligned}
$$

We associate to each specification S two programs: the local behaviour function beh_S, and the canonical specified object, obj_S. The local behaviour corresponding to S is purely functional. It is a closure with local parameters corresponding to those of the specification. When applied to a message, the behaviour function corresponding to the updated local parameters is returned along with the reply to the message. If there is shared behaviour then the current state of the shared behaviour must be passed as an argument along with the message proper, and the updated shared behaviour must be returned as well. The object specified by S has the local parameters stored in its local memory. When applied to a message, the object updates the local parameter memory and returns only the reply.

Definition (beh_S)

$$
\begin{aligned}
& beh_S(\bar{x})(msg) \leftarrow \\
& \quad \mathsf{cond}[t_0(msg, \bar{x}) \Rightarrow \langle beh_S(u_{0,1}(msg, \bar{x}), \ldots, u_{0,k}(msg, \bar{x})), r_0(msg, \bar{x})\rangle \\
& \qquad \ldots \\
& \qquad t_m(msg, \bar{x}) \Rightarrow \langle beh_S(u_{m,1}(msg, \bar{x}), \ldots, u_{m,k}(msg, \bar{x})), r_m(msg, \bar{x})\rangle \\
& \qquad \mathsf{T} \Rightarrow \langle beh_S(\bar{x}), \mathsf{nil}\rangle \\
& \qquad].
\end{aligned}
$$

Definition (obj_S)

$$
\begin{aligned}
& obj_S(\bar{z})(msg) \leftarrow \\
& \quad \mathsf{let}\{x_1 := get(z_1)\} \ldots \mathsf{let}\{x_k := get(z_k)\} \\
& \quad\; \mathsf{cond}[t_0(msg, \bar{x}) \Rightarrow \mathsf{seq}(set(z_1, u_{0,1}(msg, \bar{x})), \\
& \qquad\qquad\qquad\qquad \ldots, \\
& \qquad\qquad\qquad\qquad set(z_k, u_{0,k}(msg, \bar{x})), \\
& \qquad\qquad\qquad\qquad r_0(msg, \bar{x})) \\
& \qquad \ldots \\
& \qquad t_m(msg, \bar{x}) \Rightarrow \mathsf{seq}(set(z_1, u_{m,1}(msg, \bar{x})), \\
& \qquad\qquad\qquad\qquad \ldots, \\
& \qquad\qquad\qquad\qquad set(z_k, u_{m,k}(msg, \bar{x})), \\
& \qquad\qquad\qquad\qquad r_m(msg, \bar{x})) \\
& \qquad \mathsf{T} \Rightarrow \mathsf{nil} \\
& \qquad].
\end{aligned}
$$

There is a protocol transforming operation *beh2obj* (behaviour-to-object) that maps the behaviour corresponding to S to the object specified by S. *beh2obj* allocates a cell and stores the behaviour function there. When applied to a message it looks up the behaviour, applies it to the message, stores the new behaviour, and returns the reply. (There is also an inverse operation, but that is not needed here.) Behaviour functions and objects generalize the notions of reusable and onetime streams. The reason for having two forms is that one can often compose behaviours and reason about them more easily than the corresponding objects. Using the connections established by the abstract specification and the protocol transformation one can obtain objects corresponding to transformed behaviours. The point is that different representations are better suited for carrying out different sorts of transformations, and one needs to have appropriate representations at hand and be able to move from one representation to another in a semantically sound manner.

Definition (*beh2obj*)

$$beh2obj(beh) \leftarrow beh2objx(mk(beh))$$

$$beh2objx(z) \leftarrow \lambda(msg)\,\mathsf{let}(\langle beh, r\rangle := get(z)(msg)\}\,\mathsf{seq}(set(z, beh), r).$$

The relation between objects and behaviours, corresponding to the same specification, is captured by the following theorem.

Theorem (*beh2obj*)

$$beh2obj(beh_S(\bar{x})) \simeq \mathsf{let}(z_1 := mk(x_1)\}\ldots\mathsf{let}\{z_k := mk(x_k)\}\,obj_S(\bar{z}).$$

Proof (*beh2obj*)

The proof is by simulation induction. The object correspondence is the following:

$$\Gamma^b_{\bar{x}} = \mathsf{let}\{z := mk(beh_S(\bar{x}))\}\,[\![\,]\!]$$

$$\Gamma^o_{\bar{x}} = \mathsf{let}\{z_1 := mk(x_1)\}\ldots\mathsf{let}\{z_k := mk(x_k)\}\,[\![\,]\!]$$

$$\Gamma^b_{\bar{x}}; beh2objx(z) \simeq \Gamma^o_{\bar{x}}; obj_S(\bar{z}).$$

To verify this is a simulation correspondence we only need to show that for any *msg* we can find r, $\bar{y}$ such that

$$\Gamma^b_{\bar{x}}; beh2objx(z)(msg) \simeq \Gamma^b_{\bar{y}}; r$$

$$\Gamma^o_{\bar{x}}; obj_S(\bar{z})(msg) \simeq \Gamma^o_{\bar{y}}; r.$$

This is easily verified by using the rules for reduction. □

6 Relating notions of equivalence and fragments

Since both operational equivalence and strong isomorphism are relations defined relative to a class of contexts, it is of interest to compare these relations for various fragments of the language. We consider three such fragments: the zero-order fragment, the first-order fragment, and the full higher-order language. The zero-order fragment is built up from variables and constants using the if and let constructs together with applications of primitive operations. This fragment is studied elsewhere (Mason and Talcott, 1989*a*, *c*, to appear *a*), and a decision procedure is given for strong isomorphism.

Definition ($\mathbb{U}_{zo}\,\mathbb{E}_{zo}$)

$$\mathbb{U}_{zo} = \mathbb{X} \cup \mathbb{A}$$

$$\mathbb{E}_{zo} = \mathbb{U}_{zo} \cup \mathsf{let}\{\mathbb{X} := \mathbb{E}_{zo}\}\,\mathbb{E}_{zo} \cup \mathsf{if}(\mathbb{E}_{zo}, \mathbb{E}_{zo}, \mathbb{E}_{zo}) \cup \bigcup_{n\in\mathbb{N}} \mathbb{F}_n(\mathbb{E}^n_{zo}).$$

The first-order fragment is the language defined and studied in Mason (1986). It extends the zero order fragment by including the application of function variables together with functions defined by systems of first-order recursion equations. The only values are atoms and cells. In this fragment we let $\mathscr{F}_n$ be a set of n-ary function variables, for each $n \in \mathbb{N}$.

Definition ($\mathbb{U}_{fo}\,\mathbb{E}_{fo}$)

$$\mathbb{U}_{fo} = \mathbb{X} \cup \mathbb{A}$$

$$\mathscr{D} = \bigcup_{n\in\mathbb{N}} \langle \mathscr{F}_n, \mathbb{X}^n, \mathbb{E}_{fo}\rangle$$

$$\mathbb{E}_{fo} = \mathbb{U}_{fo} \cup \mathsf{let}\{\mathbb{X} := \mathbb{E}_{fo}\}\,\mathbb{E}_{fo} \cup \mathsf{if}(\mathbb{E}_{fo}, \mathbb{E}_{fo}, \mathbb{E}_{fo}) \cup \mathsf{recdef}(\mathscr{D}^*, \mathbb{E}_{fo}) \cup \bigcup_{n\in\mathbb{N}} (\mathscr{F}_n \cup \mathbb{F}_n)(\mathbb{E}^n_{fo}).$$

The higher order fragment is the language defined in section 2.1. Thus $\mathbb{U}_{ho} = \mathbb{U}$, and $\mathbb{E}_{ho} = \mathbb{E}$. Define $\cong_{ho}$, $\cong_{fo}$, and $\cong_{zo}$ to be operational equivalence with respect to higher-order, first-order, and zero-order contexts, respectively.

Definition ($\sqsubseteq_\Delta \cong_\Delta$)
Let $\Delta \in \{\text{zo}, \text{fo}, \text{ho}\}$ then for $e_0, e_1 \in \mathbb{E}_\Delta$ we define

$$e_0 \sqsubseteq_\Delta e_1 \Leftrightarrow (\forall C \in \mathbb{C}_\Delta)(\mathrm{FV}(C[\![e_0]\!]) = \varnothing = \mathrm{FV}(C[\![e_1]\!])) \Rightarrow (\downarrow C[\![e_0]\!] \Rightarrow \downarrow C[\![e_1]\!])$$

$$e_0 \cong_\Delta e_1 \Leftrightarrow e_0 \sqsubseteq_\Delta e_1 \wedge e_1 \sqsubseteq_\Delta e_0.$$

Define $\simeq_{ho}$, $\simeq_{fo}$ and $\simeq_{zo}$ to be strong isomorphism with respect to higher-order, first-order and zero-order memory contexts and value substitutions, respectively. Note that first-order and zero-order value expressions coincide, and hence so do the respective notions of memory contexts and value substitutions.

Definition ($\simeq_\Delta$)
Let $\Delta \in \{\text{zo}, \text{fo}, \text{ho}\}$ then for $e_0, e_1 \in \mathbb{E}_\Delta$ we define $e_0 \simeq_\Delta e_1$, if for every closing $\Gamma \in \mathbb{C}_\Delta$ and σ with $\Gamma[\![e^\sigma_j]\!] \in \mathbb{E}_\Delta$ and $\mathrm{FV}(\Gamma[\![e^\sigma_j]\!]) = \varnothing$ for $j < 2$, one of the following holds:

(1) $\uparrow(\Gamma; e^\sigma_0)$ and $\uparrow(\Gamma; e^\sigma_1)$, or

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

The situation is summarized in the following theorem.

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

Theorem (*frag*)

$$
\begin{array}{ccccc}
 & \not\Leftarrow^{b} & & & \\
e_0 \cong_{\mathrm{ho}} e_1 & \Rightarrow & e_0 \cong_{\mathrm{fo}} e_1 & \Leftrightarrow & e_0 \cong_{\mathrm{zo}} e_1 \\
\not\Downarrow^{e} \Uparrow^{a} & & \Updownarrow^{c} & & \Updownarrow^{d} \\
e_0 \simeq_{\mathrm{ho}} e_1 & \Rightarrow & e_0 \simeq_{\mathrm{fo}} e_1 & \Leftrightarrow & e_0 \simeq_{\mathrm{zo}} e_1 \\
 & \not\Leftarrow^{b} & & &
\end{array}
$$

Proof (*frag*)

The horizontal implications are simple consequences of the corresponding containment relations for the relevant contexts. The implication labelled (a) is a consequence of weak extensionality (*opeq.striso*). The negated implications (b) is due essentially to type discrimination capability of the language. A counterexample is $e_0 = eq(x, x)$ and $e_1 = \mathsf{T}$. 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 $eq(\lambda x.x, \lambda x.x) \simeq \mathsf{Nil}$.† The implication (c) is a consequence of weak extensionality for the first-order fragment (*fo.ciu*), see below. The implication (d) follows from the fact that if $e_0 \simeq_{\mathrm{fo}} e_1$ does not hold then we can find a zero-order memory context Γ, value substitution σ, and reduction context R such that $\Gamma; R[\![e_0^{\sigma}]\!]$ is defined and $\Gamma; R[\![e_1^{\sigma}]\!]$ is not defined. By sufficiently unfolding any recursive function calls it is easy to see that R can be found in the zero-order fragment. An example that establishes the negated implication (e) has been given previously. For example, $\lambda x.x \cong_{\mathrm{ho}} \lambda x.\mathsf{seq}(x, x)$ but not $\lambda x.x \simeq_{\mathrm{ho}} \lambda x.\mathsf{seq}(x, x)$. □

Theorem (*fo.ciu*)

$e_0 \cong_{\mathrm{fo}} e_1$ iff for all closing Γ, σ, R we have $\Gamma; R[\![e_0^{\sigma}]\!]$ is defined iff $\Gamma; R[\![e_1^{\sigma}]\!]$ is defined.

Proof (*fo.ciu*)

The forward implication is trivial. For the backward implication assume $\Gamma; R[\![e_0^{\sigma}]\!]$ is defined iff $\Gamma; R[\![e_1^{\sigma}]\!]$ is defined. Show by computation induction that for any closing $\Gamma; C$, $\Gamma; C[\![e_0]\!]$ is defined iff $\Gamma; C[\![e_1]\!]$ is defined where C is a context with holes decorated by value substitutions. Note that in contrast to the higher-order case, in the first order case we may restrict to simple memory contexts and value substitutions without holes. Assume $\Gamma; C[\![e_0]\!]$ is defined. C is either a value, a reduction context with a redex in the reduction hole or of the form $R[\![\varepsilon^{\sigma}]\!]$. In the first case we are done trivially. In the second case $\Gamma; C$ reduces uniformly to a smaller computation without touching any holes. In the third case we use our initial assumption. □

As noted above, strong isomorphism is a stronger notion than operational equivalence for the full language, since any two operationally equivalent but distinct λ-expressions will provide a counterexample. In fact these are the only counterexamples. The following theorem states that operational equivalence and strong isomorphism coincide on a natural fragment of the full higher-order language, $\mathbb{E}_{\neg\lambda}$. This is a generalization of the theorem of Mason (1986), p 48).

† 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

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

Theorem (*foc*)
If $e_0, e_1 \in \mathbb{E}_{\neg\lambda}$ and $e_0 \cong_{\text{ho}} e_1$, then $e_0 \simeq_{\text{ho}} e_1$.

Proof (*foc*)
We begin by stating a lemma that isolates the problem at hand.

Lemma (*wk.striso*)
Suppose that $e_0 \cong e_1$ and that $\downarrow \Gamma ; e_i, \Gamma[\![e_i]\!] \in \mathbb{E}_{\varnothing}$ for $i < 2$. Then there exist Γ_i, Γ', u, σ_i for $i < 2$ such that

- $\Gamma ; e_i \overset{*}{\mapsto} \Gamma_i \cup (\Gamma' ; u)^{\sigma_i}$ where we have that $\Gamma'[\![u]\!] \in \mathbb{E}_{\neg\lambda}$
- $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$, $\mathrm{Dom}(\sigma_0) = \mathrm{Dom}(\sigma_1) = \mathrm{FV}(\Gamma'[\![u]\!])$ and $\mathrm{Rng}(\sigma_i) \subset \mathbb{L}$.

Proof (*wk.striso*)
Let $\mathrm{Dom}(\Gamma) = \{z_0, \ldots, z_n\}$ and consider the context

$$\Gamma[\![\langle \varepsilon, z_0, z_1, \ldots, z_{n-1}, z_n \rangle]\!]. \qquad \square$$

Suppose that $e_i \in \mathbb{E}_{\neg\lambda}$, $i < 2$, $e_0 \cong e_1$, and Γ, σ are such that $\Gamma[\![e_i]\!] \in \mathbb{E}_{\neg\lambda}$, $\sigma = \{x_j := \rho_j \mid j \leqslant n\}$ where $\rho_j = \lambda y . e'_j \in \mathbb{L}_{\mathrm{Dom}(\Gamma)}$ for $j \leqslant n$, $\rho_j \neq \rho_k$ whenever $j \neq k$, $\mathrm{Dom}(\sigma) = \mathrm{FV}(\Gamma[\![e_i]\!])$, and $\downarrow (\Gamma ; e_i)^\sigma$ for $i < 2$. Now by (*wk.striso*) we have that there exist Γ_i, Γ', u, σ_i for $i < 2$ such that

- $(\Gamma ; e_i)^\sigma \overset{*}{\mapsto} \Gamma_i \cup (\Gamma' ; u)^{\sigma_i}$ where $\Gamma'[\![u]\!] \in \mathbb{E}_{\neg\lambda}$
- $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$, $\mathrm{Dom}(\sigma_0) = \mathrm{Dom}(\sigma_1) = \mathrm{FV}(\Gamma'[\![u]\!])$ and $\mathrm{Rng}(\sigma_i) \subset \mathbb{L}$.

Thus the main work in proving the theorem is in showing that $\sigma_0 = \sigma_1$ and that $\mathrm{Rng}(\sigma_i) \subset \mathbb{L}_{\mathrm{Dom}(\Gamma')}$. Note that this last fact concerning the σ_i implies that $\mathrm{FV}((\Gamma' ; u)^{\sigma_i}) = \varnothing$, in other words that Γ_i is garbage. We illustrate a simple case in detail and then sketch the general case.

Simple case
Suppose that no pfns are created in the course of evaluating either $(\Gamma ; e_0)^\sigma$ or $(\Gamma ; e_1)^\sigma$.

Proof (*simple case*)
In this case we have that $\mathrm{Rng}(\sigma_i) \subseteq \mathrm{Rng}(\sigma)$ for $i < 2$ and so we have that $\mathrm{Rng}(\sigma_i) \subset \mathbb{L}_{\mathrm{Dom}(\Gamma')}$. Thus in this case we need only show that $\sigma_0 = \sigma_1$. Now pick entirely new variables $z_0, \ldots, z_n$ and $w_0, \ldots, w_n$ and put

$$\begin{aligned} \rho'_j &= \lambda y . \mathsf{if}(eq(y, z_j), w_j, e'_j) \\ \sigma' &= \{x_j := \rho'_j \mid j \leqslant n\} \\ \Gamma_2 &= \{z_j := \langle \mathsf{T} \rangle, w_j := \langle \mathsf{T} \rangle \mid j \leqslant n\}. \end{aligned}$$

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press

Then by construction the respective computations are essentially unchanged and therefore

$$\Gamma_2 \cup (\Gamma; e_\iota)^{\sigma'} \overset{*}{\mapsto} \Gamma_2 \cup \Gamma_\iota \cup (\Gamma'; u)^{\sigma'_\iota}$$

where $\sigma'_\iota(x) = \rho'_j \Leftrightarrow \sigma_\iota(x) = \rho_j$. Now suppose that $\sigma'_0(x) = \rho'_j$ and $\sigma'_1(x) \neq \rho'_j$. Then by construction

$$\Gamma_2 \cup (\Gamma; \mathsf{seq}(e_0, eq(\mathsf{app}(x, z_j), w_j)))^{\sigma} \overset{*}{\mapsto} \Gamma_2 \cup \Gamma_0 \cup (\Gamma'; \mathsf{T})^{\sigma'_0}$$

$$\Gamma_2 \cup (\Gamma; \mathsf{seq}(e_1, eq(\mathsf{app}(x, z_j), w_j)))^{\sigma} \overset{*}{\mapsto} \Gamma_2 \cup \Gamma_1 \cup (\Gamma'; \mathsf{Nil})^{\sigma'_1}.$$

Which together with (*congruence*) contradicts the assumption that $e_0 \cong e_1$. $\square$

General case

In this case we allow new pfns to be created in the course of evaluating either $(\Gamma; e_0)^\sigma$ or $(\Gamma; e_1)^\sigma$. This case requires substantially more work than in the previous case, although the idea is essentially the same. We must enclose each pre-existing pfn in a shell which performs a substantial amount of bookkeeping. Just as in the simple case the envelope will, when queried, reveal the identity of the pfn. It will also keep track of the progress of computation, and add to any newly created pfn a similar encasing. As a result of this extra work we obtain much more information concerning the relationship between e_0 and e_1. We begin by defining a bookkeeping pfnl, *trace*, using the standard notation for recursive definition. The definition has five free variables y_0, y_1, y_2, y_3, y_4, which will refer to lists that will be used to store information. We shall describe their purpose and contents in detail after we have defined *trace*.

$$
\begin{array}{l}
trace(z, w, x_\rho, x) \leftarrow \\
\quad \mathsf{if}(eq(x, z), \\
\qquad w, \\
\qquad \mathsf{seq}(nconc(y_2, cells(x)) \\
\qquad\quad \mathsf{let}\{c_1 := \langle z, record(\langle x, y_2 \rangle)\rangle, \\
\qquad\qquad v := \mathsf{app}(x_\rho, x)\} \\
\qquad\qquad \mathsf{cond}[atom(v) \Rightarrow v \\
\qquad\qquad\qquad cell(v) \Rightarrow \mathsf{seq}(nconc(y_2, cells(v)), mapcar(tracecell, y_2), v) \\
\qquad\qquad\qquad \mathsf{T} \Rightarrow \mathsf{seq}(mapcar(tracecell, y_2), tracepfn(v))])) \\
tracepfn(x_\rho) \leftarrow \\
\quad \mathsf{let}\{c_3 := \langle \mathsf{T} \rangle, c_4 := \langle \mathsf{T} \rangle\} \\
\qquad \mathsf{seq}(nconc(y_0, \langle trace(c_3, c_4, x_\rho)\rangle), nconc(y_3, c_3), nconc(y_4, c_4), \\
\qquad\quad trace(c_3, c_4, x_\rho)) \\
tracell(x) \leftarrow \\
\quad \mathsf{if}(atom(x), \\
\qquad x, \\
\qquad \mathsf{let}\{x_a := car(x), x_d := cdr(x)\}
\end{array}
$$

$$\begin{aligned}\mathsf{cond}[\mathsf{and}(pfn(x_a), pfn(x_d)) &\Rightarrow \mathsf{seq}(setcar(x, tracepfn(x_a)),\\ &\qquad setcdr(x, tracepfn(x_d)),\\ &\qquad x)\\ pfn(x_a) &\Rightarrow \mathsf{seq}(setcar(x, tracepfn(x_a)), x)\\ pfn(x_d) &\Rightarrow \mathsf{seq}(setcdr(x, tracepfn(x_d)), x)\\ \mathsf{T} &\Rightarrow x])\end{aligned}$$

$$pfn(x) \Leftarrow \mathsf{not}(\mathsf{or}(atom(x), cell(x))).$$

We begin by describing the intended values of the arguments and global parameters, we then define the four auxiliary pfns, *nconc*, *mapcar*, *cells* and *record*.

Just as in the simple case, two newly created cells will be the values of the parameters z and w. They are used as *indicators* or *signatures*, in the sense that each pfn will reveal its identity uniquely via these two cells. When applied to the cell z the traced pfn will return the cell w. The cell z is called the *query* indicator, while w is called the *reply* indicator. As pfns are created, and consequently traced, more of these indicators will be allocated. To keep a record of each *query* and its corresponding *reply* indicator, the two lists y_3 and y_4 come into play. the list y_3 stores all the *query* indicators, while the list y_4 stores the corresponding *reply* indicators. The list y_0 contains all the accessible pfns, both pre-existing, as well as those created in the course of the computation. This list may contain many duplications, it will however be exhaustive. The list y_1 contains a list of pairs. Each pair corresponds to the application of a pfn to an argument. The first element of the pair is the *query* indicator of the pfn, while the second argument is a persistent record of the argument to the pfn as well as the state at the time of application. This persistent record is constructed via the auxiliary program *record*, which we will say more about shortly. the list y_2 contains a list of all the cells that have been accessible at one time or another in the computation. As in the case of the list of pfns, duplication is sacrificed for exhaustiveness. It is used each time a pfn is applied, as a means of recording the state at the time of application.

nconc is the usual destructive list operation. It destructively appends its second argument onto the tail of its first, both being lists. *mapcar* is the usual Lisp mapping operation. It applies its first argument to each element of its second argument, *cons*es up a list of results and returning it as the value. *cells*, simply returns a list of all those cells reachable from its argument, in left-first order say.

The definition of the third, *record*, is quite complex, unlike its function which is to record, persistently, the structure of its argument. In other words we wish *record*(x) to return an entity which stores or records the structure of x *at the time of application*. This entity should be insensitive to any possible later modifications to x. One solution is that *record*(x) should return a pfn that behaves like the path function of x (at the time of application). In short, *record* is a simple programming problem, and we specify its assumed behaviour leaving its definition as an exercise.

Definition (*record*)

$(\forall\Gamma[\![x]\!]\in\mathbb{E}_{\varnothing})(\exists\Gamma_{\rho_x},\rho_x)$ such that $\Gamma;record(x)\mapsto\Gamma_{\rho_x}\cup\Gamma;\rho_x$ and for any Γ' with $\mathrm{Dom}(\Gamma)\subseteq\mathrm{Dom}(\Gamma')$ and $\mathrm{Dom}(\Gamma')\cap\mathrm{Dom}(\Gamma_{\rho_x})=\varnothing$ and any *car-cdr* chain

$$\Theta=\vartheta_0(\vartheta_1(\ldots\vartheta_k(\varepsilon)\ldots))$$

where $\vartheta_j\in\{car,cdr\}$, $j\leqslant k$, we have that

$$(\Gamma_{\rho_x}\cup\Gamma';\rho_x(\lambda z.\Theta[\![z]\!])\mapsto\Gamma_{\rho_x}\cup\Gamma';u)\Leftrightarrow(\Gamma;\Theta[\![x]\!]\mapsto\Gamma;u).$$

Suppose $\mathrm{Dom}(\Gamma)=\{c_0,\ldots,c_s\}$. Now as in the simple case pick entirely new variables $z_0,\ldots,z_n$, $w_0,\ldots,w_n$ and $y_0,\ldots,y_4$ and put

$$\begin{aligned}
\rho_j' &= trace(z_j,w_j,y_0,y_1,y_2,y_3,y_4,\rho_j)\\
\sigma' &= \{x_j:=\rho_j' \mid j\leqslant n\}\\
\Gamma_{trace} &= \{z_j:=\langle\mathsf{T}\rangle\\
&\qquad w_j:=\langle\mathsf{T}\rangle\\
&\qquad y_0:=\langle x_0,\ldots,x_n\rangle\\
&\qquad y_1:=\langle\mathsf{T}\rangle\\
&\qquad y_2:=\langle c_0,\ldots,c_s\rangle\\
&\qquad y_3:=\langle z_0,\ldots,z_n\rangle\\
&\qquad y_4:=\langle w_0,\ldots,w_n\rangle \mid j\leqslant n\}.
\end{aligned}$$

Then again by construction the respective computations are essentially unchanged (modulo the additional bookkeeping being done), and therefore

$$(\Gamma_{trace}\cup\Gamma;e_i)^{\sigma'}\mapsto\Gamma_i\cup(\Gamma_{trace}^i\cup\Gamma';u)^{\sigma_i'}.$$

Also, by considering the context

$$(\Gamma_{trace}\cup\Gamma;\langle\varepsilon,y_0,y_1,y_2,y_3,y_4\rangle)^{\sigma'}$$

we can require that $\Gamma^0_{trace}=\Gamma^1_{trace}$. Now the resulting bookkeeping Γ^i_{trace} contains, amongst other things, the five lists y_0, y_1, y_2, y_3 and y_4. The first list y_0 is (under the substitution σ_i') a list of all the pfns both already existing and newly created. Consequently we may assume that $\mathrm{Dom}(\sigma_0')=x_0,\ldots,x_n,x_0',\ldots,x_m'=\mathrm{Dom}(\sigma_1')$ and that $\Gamma'_{trace}(y_0)=\langle x_0,\ldots,x_n,x_0',\ldots,x_m'\rangle$. The presence of $z_0,\ldots,z_n$ and $w_0,\ldots,w_n$ in Γ_{trace} forces $\sigma_0'(x_i)=\sigma_1'(x_i)$ for $i\leqslant n$. Consequently, we need only show that $\sigma_0'(x_i')=\sigma_1'(x_i')$ for $i\leqslant m$ and we are done. Suppose that $\sigma_0'(x_0')=\rho_a$ *and* $\sigma_1'(x_i')=\rho_b$, then by looking at the very first pair stored in the list y_1, we see that these pfns were created by applying the very same pfn to the very same argument in the very same state. (Here is where the persistency of *record* is needed.) Thus we may assume that $\rho_a=\rho_b$. Continuing this line of reasoning yields the desired conclusion. □

7 Conclusions

The results presented in this paper provide basic tools for specifying and reasoning about objects with memory and about programs acting on such objects. Our language is close to existing applicative (functional) languages such as Lisp, Scheme and ML. An important feature is that 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 weak extensionality characterization of operational approximation and equivalence (*ciu*). This is the basis of several important methods for proving approximation and equivalence. (*ciu*) extends the *safety* theorem of Felleisen (1987, Theorem 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 cannot express non-termination we have that safe equivalence implies operational equivalence but not conversely.

The key point of the proof methods based on (*ciu*) is that they reduce the problem to reasoning about computations where we can argue by cases and computation induction. What is needed now is to determine a small collection of rules that comprise the main uses of computation induction and to develop further syntactic methods for conditional reasoning. One approach is to extend the constraint techniques used for the first-order completeness result in Mason and Talcott (1989*a*, *c*, to appear *a*). Several examples that illustrate our techniques for reasoning about programs with effects are given elsewhere (Mason and Talcott, 1990). They include the following: introducing a parameter to make single threaded store explicit; moving expressions that effect common structure together and simplifying to express the cumulative effect; moving an expression describing the computation of a value closer to its point of use (possibly modifying the description to make the move valid); representing mutable structure in abstract objects to encapsulate effects and potential interference in a controlled way and to maintain invariants and representation integrity; and formulating induction principles that are valid in the presence of effects. Progress towards a theory of program development by systematic refinement is described elsewhere (Mason and Talcott, to appear *c*). Here a formal system for propagating constraints into program contexts is presented. In this system, it is possible to place expressions equivalent under some non-empty set of contraints into a program context and preserve equivalence provided that the constraints propagate into that context. Constrained equivalence and constraint propagation provide a basis for systematic development of program transformation rules. Three key rules are: subgoal induction, recursion induction, and the peephole rule. We report progress in development of methods for reasoning about the equivalence of objects with memory and the use of these methods to describe sound operations on such objects, in terms of formal program transformations in (Mason and Talcott, to appear *b*). We also formalize three different aspects of objects: their specification, their behaviour, and their canonical representative. Formal connections among these aspects provide methods for optimization and reasoning about systems of objects. To illustrate these ideas we give a formal derivation of an optimized specialized window editor from generic specifications of its components. A new result based on simulation induction is presented that enables one to make use of symbolic evaluation (with respect to a set of constraints) to establish the equivalence of objects.

Talcott (1985) defines a class of pre-orderings called comparison relations and suggests maximal comparisons as an alternative to the methods of Scott (see Barendregt, 1981, Chapter 3) for obtaining extensional models of lambda calculi. Talcott (1989) shows that for a language with function and control abstractions operational approximation as traditionally defined is not a comparison relation. A refinement of operational equivalence is defined and shown to be the maximum comparison relation. This equivalence relation is the basis of a fully quantified equational theory of function and control abstractions, and many examples of properties of programs are stated and proved. In Talcott (1990) this work is formalized within a logic of variable types (Feferman, 1985, 1990).

Abramsky (1990) introduces notions of applicative transition system and bisimulation relation to provide meaning for lambda terms appropriate for lazy evaluation. Domain theory is used to characterize the maximum bisimulation relation and to prove full abstraction results. Howe (1989) introduces a notion of a lazy computation system that provides a richer term language than that of lambda transition systems and extends the notion of bisimulation relation to this case. A technique of extension by closure conditions is used to prove that the maximum bisimulation is a (pre)congruence. This is similar to methods used in Talcott (1985) to reason about comparison relations. Smith (1991) applies the methods of Howe to develop syntactic notions analogous to the domain theoretic notions of least-upper and greatest-lower bound. These are used to prove the least-fixed-point property of the Y combinator and to develop a computation induction principle. In cases where bisimulation and operational approximation agree, it appears that methods similar to those used in Talcott (1985, 1989), and in the present work yield simpler proofs of a number of theorems (S. Smith, private communication). However, bisimulation provides an alternative approach to equivalence and deserves consideration in computation systems that permit effects other than non-termination. The definition of bisimulation relation assumes that extensionality is consistent. Since the presence memory effects makes this no longer true, the basic definition would require some modification in order to extend the methods of Abramsky and Howe to the computational language presented in this paper. We plan to investigate this approach.

An early effort in the direction of equational theories for proving correctness of higher-order imperative programs is Demers and Donahue (1983). 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 three unary and one binary relation in their system. The unary relations express the legality, well-formedness and purity of expressions, while the binary relation represents some intensional form of equivalence. The simultaneous deduction of legality, well-formedness, purity and equivalence makes the rules very complex. There are no formal results on the equational theory nor its relationship to the original lambda calculus. Boehm (1985) defines a first-order theory for reasoning about programs in the language Russell. Program constructs are defined by two classes of axioms: (1) axioms about the value returned; and (2) axioms giving the effect on memory. Some relative completeness results are given, but no decidable fragments are considered. Implicit in the completeness result of Mason and Talcott (1989*a*, *c*) is a decision procedure for the semantic consequence relation. This is an important step towards developing computer-aided deduction tools for reasoning about programs with memory. This extended the work of Oppen (1978), which gives a decision procedure for the first-order theory of pure Lisp, i.e. the theory of *atom*, *car*, *cdr*, *cons* over acyclic list structures. Nelsen and Oppen (1977) treat the quantifier-free case over possibly cyclic list structures, but neither treats updating operations.

Notions of effect and interference are used informally (Mason and Talcott, in preparation) to give intuitive explanations of technical properties and results. These notions are not new. Reynolds (1989) gives purely syntactic criteria for avoiding interference. Rather than prohibit interference entirely the aim is to isolate occurrences of interference and to make them syntactically obvious. This is accomplished by requiring that interference occur only within object like entities. This is very similar in spirit to our use of abstract objects to encapsulate access to structures. Our motivation is to be able to use this abstraction to facilitate reasoning about programs. Gifford and Lucassen (1987, 1988) formalize notions of read, write and allocate effects for a language very similar to ours. An inference system for deducing effect types is defined and based on this system criteria are given for determining when expressions interfere, when results can be cached rather than being recomputed, etc. These methods should be contrasted with the more restrictive approaches that have recently been proposed. In Wadler (1990) a type system using linear logic is used to enforce the *single-threadedness* of mutated objects. A similar goal is achieved by somewhat different syntactic means in Guzmán and Hudak (1990) and Reddy *et al.* (1990). We expect that combining the work on effect and interference with the work on program equivalence will provide much more powerful tools for reasoning about programs as well as increasing the utility of the effect systems for automatic manipulation of programs.

Acknowledgements

The authors would like to thank both Lou Galbiati and Furio Honsell for numerous helpful discussions and careful reading of previous versions. We also wish to thank the two anonymous referees for many helpful comments and criticisms. This research was partially supported by DARPA contract N00039-84-C-0211 and NSF grants CCR-8718605, CCR-8915663.

Appendix: index of notations

Symbol	Description	section
$\mathbb{N}$	The natural numbers, $i, j, \ldots, n \in \mathbb{N}$	1
Y^n	Sequences of elements of Y of length n	1
Y^*	Finite sequences of elements of Y	1
$P_\omega(Y)$	Finite subsets of Y	1
$Y_0 \to Y_1$	Total functions from Y_0 to Y_1	1
$\mathrm{Dom}(f)$	The domain of the function f	1
$\mathrm{Rng}(f)$	The range of the function f	1
$f\{y := y'\}$	An extension to, or alteration of, the function f	1
$\mathbb{X}$	A countably infinite set of variables	2.1
x, y, z	Generic elements of $\mathbb{X}$	2.1
$\mathbb{A}$	The set of atoms	2.1
a	Generic element of $\mathbb{A}$	2.1
T, Nil	Atoms playing the role of booleans	2.1
$\mathbb{F}_1$	Unary memory operations $\supseteq \{atom, cell, car, cdr\}$	2.1
$\mathbb{F}_2$	Binary memory operations $\supseteq \{eq, cons, setcar, setcdr\}$	2.1
$\mathbb{F}_n$	the set of n-ary operation symbols	2.1
$\mathbb{F}$	The set of all operation symbols	2.1
$\lambda x.e$	A lambda abstraction, also called a *pfn*	2.1
$\mathbb{L}$	The set of λ-abstractions	2.1
ρ	Generic elements of $\mathbb{L}$	2.1
$\mathbb{U}$	The set of value expressions	2.1
u	Generic element of $\mathbb{U}$	2.1
$\mathbb{E}$	The set of expressions	2.1
e	Generic element of $\mathbb{E}$	2.1
$\mathsf{if}(e_0, e_1, e_2)$	Conditional branching	2.1
$\mathsf{app}(e_0, e_1)$	Application	2.1
$\delta(\bar{e})$	Application of primitive operations	2.1
$\mathrm{FV}(e)$	The free variables of the expression e	2.1
$\mathbb{E}_\varnothing$	The set of expressions with no free variables	2.1
$e\{x := e'\}$	The result of substituting e' for x in e	2.1
σ	A value substitution	2.1
e^σ	The result of carrying out the substitution σ	2.1
ε	The hole used to make contexts	2.1
$\mathbb{C}$	The set of contexts	2.1
C	Generic element of $\mathbb{C}$	2.1
$C[\![e]\!]$	The result of filling the context with e	2.1
$\lambda x_1, \ldots, x_n . e$	n-ary lambda abstraction	2.1
$e_0(e_1, \ldots, e_n)$	n-ary function application	2.1
$\mathsf{let}\{x := e_0\} e_1$	Lexical variable binding	2.1
$\mathsf{seq}(e_1, \ldots, e_n)$	Sequencing construct	2.1
$\mathsf{cond}[\ldots, e_i \Rightarrow e'_i, \ldots]$	The Lisp conditional	2.1
mk	Unary cell construction	2.1
get	Unary cell access	2.1
set	Unary cell updating	2.1
$\langle u_1, \ldots, u_n \rangle$	The S-expression list with elements u_i	2.1
$\mathbb{E}_{\mathsf{redex}}$	The set of redexes	2.2
$\mathbb{R}$	The set of reduction contexts	2.2
R	Generic element of $\mathbb{R}$	2.2
$\mathbb{M}$	The set of memory contexts	2.2

Appendix: index of notations (cont.)

Symbol	Description	section
Γ	Generic element of $\mathbb{M}$	2.2
$\mathbb{D}$	The set of descriptions	2.2
$\Gamma; e$	Generic element of $\mathbb{D}$	2.2
$\Gamma; u$	Value description	2.2
$\Gamma; \rho$	Pfn object	2.2
$\xrightarrow{p}$	The primitive reduction relation on $\mathbb{D} \times \mathbb{D}$	2.2
$\mapsto$	The single step reduction relation on $\mathbb{D} \times \mathbb{D}$	2.2
$\overset{*}{\mapsto}$	The reduction relation on $\mathbb{D} \times \mathbb{D}$	2.2
$\downarrow$	The *is defined* predicate on descriptions	2.2
$\uparrow$	The *is not defined* predicate on descriptions	2.2
$(\Gamma; e)^{\sigma}$	Substitution into a description	2.2
$\sqsubseteq$	Operational approximation	3.0
$\cong$	Operational equivalence	3.0
$\sqsubseteq^{0}$	Trivial approximation	3.0
$\sqsubseteq^{ciu}$	All closed instances of all uses are approximate	3.1
$\sqsubseteq^{ci}$	All closed instances are approximate	3.1
$\simeq$	Strong isomorphism	3.2
rec	Recursion operator	4.0
$f(\bar{x}) \leftarrow e$	Recursive function definition	4.0
rec_v	The call-by-value fixed-point combinator	4.1
rec_m	The *cyclic* fixed point combinator	4.1
$\mathcal{O}$	An object correspondence	5.0
$\mathbb{U}_{zo}$	Zero order value expressions	6.0
$\mathbb{E}_{zo}$	Zero order expressions	6.0
$\cong_{zo}$	Operational equivalence w.r.t. the zero order fragment	6.0
$\simeq_{zo}$	Strong isomorphism w.r.t. the zero order fragment	6.0
$\mathbb{U}_{fo}$	First order value expressions	6.0
$\mathbb{E}_{fo}$	First order expressions	6.0
$\cong_{fo}$	Operational equivalence w.r.t. the first order fragment	6.0
$\simeq_{fo}$	Strong isomorphism w.r.t. the first order fragment	6.0
$\mathbb{U}_{ho}$	Higher order value expressions	6.0
$\mathbb{E}_{ho}$	Higher order expressions	6.0
$\cong_{zo}$	Operational equivalence w.r.t. the higher order fragment	6.0
$\simeq_{zo}$	Strong isomorphism w.r.t. the higher order fragment	6.0
$\mathbb{E}_{\neg\lambda}$	The set of λ-free expressions	6.0

References

Abramsky, S. 1990. The lazy lambda calculus. In D. A. Turner (editor), *Research Topics in Functional Programming*. Addison-Wesley.

Agha, A. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

Barendregt, H. 1981. *The lambda calculus: its syntax and semantics*. North-Holland.

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

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

Feferman, S. 1985. A theory of variable types. *Revista Colombiana de Matématicas*, **19**.

Feferman, S. 1990. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In

Logic and Computation (*vol. 104*) *Contemporary Mathematics*. A.M.S.
Felleisen, M. 1987. *The Calcului of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, USA.
Felleisen, M. and Hieb, R. 1989. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, USA.
Guzmán, J. C. and Hudak, P. 1990. Single-threaded polymorphic lambda calculus. In *Fifth Ann. Symposium on Logic in Computer Science*. IEEE Press.
Hewitt, C. 1977. Viewing control structures as patterns of passing messages. *J. Artificial Intelligence*, **8** (3).
Howe, D. 1989. Equality in the lazy lambda calculus. In *Fourth Ann. Symposium on Logic in Computer Science*. IEEE Press.
McCarthy, J., Abrahams, P., Edwards, D., Hart, T. and Levin, M. 1962. *Lisp 1.5 Programmer's Manual*. MIT Press.
Landin, P. J. 1964. The mechanical evaluation of expressions. *Computer J.*, **6**: 308–320.
Lucassen, J. M. 1987. *Types and Effects, towards the integration of functional and imperative programming*. PhD thesis, MIT, USA.
Lucassen, J. M. and Gifford, D. K. 1988. Polymorphic effect systems. In *Conf. record 16th Ann. ACM symposium on principles of programming languages*, 47–57, ACM.
Mason, I. A. 1986. *The Semantics of Destructive Lisp*. PhD thesis, Stanford University, USA.
Mason, I. A. 1988. Verification of programs that destructively manipulate data. *Sci. of Computer Programming*, **10**.
Mason, I. A. and Talcott, C. L. 1989*a*. Axiomatizing operational equivalence in the presence of side effects. In *Fourth Ann. Symposium on Logic in Computer Science*. IEEE Press.
Mason, I. A. and Talcott, C. L. 1989*b*. Programming, transforming, and proving with function abstractions and memories. In *Proc. 16th EATCS Colloquium on Automata, Languages, and Programming*, Stresa, Italy. Volume 372 of *Lecture Notes in Computer Science*, Springer-Verlag.
Mason, I. A. and Talcott, C. L. 1989*c*. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Department of Computer Science, Stanford University, USA.
Mason, I. A. and Talcott, C. L. Inferring the equivalence of functional programs that mutate data. To appear (*a*) in *Theor. Comput. Sci.*
Mason, I. A. and Talcott, C. L. Syntactic semantics. In preparation.
Mason, I. A. and Talcott, C. L. Program transformation for configuring components. To appear (*b*) ACM Sigplan Symposium on Partial Evaluation and Semantics Based Program Manipulation. PEPM91.
Mason, I. A. and Talcott, C. L. Program transformation via constraint propagation. To appear (*c*).
Mason, I. A. and Talcott, C. L. 1990. Reasoning about programs with effects. In *Programming Language Implementation and Logic Programming, PLILP'90*. Volume 456 of *Lecture Notes in Computer Science*, Springer-Verlag.
Moggi, E. 1989. Computational lambda-calculus and monads. In *Fourth Ann. Symposium on Logic in Computer Science*. IEEE Press.
Moggi, E. 1990. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for foundations of computer science, University of Edinburgh, UK.
Morris, J. H. 1968. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, USA.
Mosses, P. 1984. A basic abstract semantic algebra. In *Semantics of data types, international symposium*, Sophia-Antipolis, France. Volume 173 of *Lecture Notes in Computer Science*, Springer-Verlag.
Nelson, C. G. and Oppen, D. C. 1977. Fast decision procedures based on congruence closure. Technical Report STAN-CS-77-647, Department of Computer Science, Stanford University, USA.

Oppen, D. C. 1978. Reasoning about recursively defined data structures. Technical Report STAN-CS-78-678, Department of Computer Science, Stanford University, USA.
Plotkin, G. 1975. Call-by-name, call-by-value and the lambda-v-calculus. *Theor. Comput. Sci.*, 1.
Reddy, U., Swarup, V. and Ireland, E. 1990. Assignments for applicative languages. Unpublished Manuscript, Department of Computer Science, University of Illinois, USA.
Reynolds, J. C. 1972. Definitional interpreters for higher-order programming languages. In *Proc. ACM Nat. Convention*, ACM.
Reynolds, J. C. 1989. Syntactic control of interference, II. *Proc. 16th EATCS Colloquium on Automata, Languages, and Programming*, Stresa, Italy. Volume 372 of *Lecture Notes in Computer Science*, Springer-Verlag.
Smith, S. F. From operational to denotational semantics. Technical Report 89-12, Department of Computer Science, The Johns Hopkins University, USA. To appear in the Proceedings of the 7th Conference on the Mathematical Foundations of Programming Semantics, MFPS91 LNCS Springer Verlag.
Steele, G. L. and Sussman, G. J. 1975. Scheme, an interpreter for extended lambda calculus. Technical Report 349, Massachusetts Institute of Technology, USA.
Talcott, C. L. 1985. *The Essence of Rum: A Theory of the Intensional and Extensional Aspects of Lisp-Type Computation*. PhD thesis, Stanford University, USA.
Talcott, C. L. 1989. Programming and proving function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department, USA.
Talcott, C. L. 1990. A theory for program and data specification. *Design and Implementation of Symbolic Computation Systems, DISCO'90*. Volume 429 of *Lecture Notes in Computer Science*, Springer-Verlag.
Wadler, P. 1990. Linear types can change the world! In *IFIP Working Conf. Programming Concepts and Methods*, Sea of Gallilee, Israel.
Yonezawa, A. (editor). 1990. *ABCL: An Object-Oriented Concurrent System*. MIT Press.

https://doi.org/10.1017/S0956796800000125 Published online by Cambridge University Press