Reasoning about programs with effects

Ian Mason
Stanford University
IAM@SAIL.STANFORD.EDU

Carolyn Talcott
Stanford University
CLT@SAIL.STANFORD.EDU

1. Introduction

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. To accomplish this it is necessary to identify structures and constructs that provide expressive power and at the same time facilitate mathematical reasoning. Here we focus on a language with function abstractions and operations for creating, accessing, and modifying memory. In this setting features of algorithmic, functional, and object-oriented language can be treated. Central to our approach is the study of various useful notions of program equivalence. Notions of program equivalence are fundamental for the process of program specification, derivation, and transformation, as well as other forms of code generation and optimization.

In this paper we give several examples that illustrate our techniques for reasoning about programs with effects. Techniques illustrated include the following: (i) removing inessential (non-visible) effects such as replacing assignment to local variables by let-binding; (ii) introducing a parameter to make single threaded store explicit; (iii) moving expressions that effect common structure together and simplifying to express the cumulative effect; (iv) moving an expression describing the computation of a value closer to its point of use (possibly modifying the description to make the move valid); (v) representing mutable structure in abstract objects to encapsulate effects and potential interference in a controlled way and to maintain invariants and representation integrity; and (vi) formulating induction principles that are valid in the presence of effects.

The remainder of the paper is organized as follows. In section 2. we review the syntax and semantics of our language, define notions of program equivalence and state some key theorems. In section 3. we give a simple example illustrating the benefits of functional style programming for defining operations with effects. This illustrates the removal of inessential effects. In section 4. we show how structure copying and structural induction can be used to prove properties of programs. Structure copying is a method of moving descriptions of values closer to the point of use. In section 5. we introduce the notion of an abstract object and illustrate the use of abstract objects for specifying and reasoning about programs. In section 6. we discuss some additional ideas and related work that will help to further ease the burden of understanding program behavior. We generally use Common Lisp names for functions and do not include these definitions in the text, they are collected in an appendix for completeness.

2. Computation over Memory Structures.

Formally our language is an extension of the call-by-value lambda calculus obtained by adding primitive operations that create, access, and modify memory cells (together with a collection of basic constants and operations on these basic constants). Our language can be thought of as untyped ML or as a variant of Scheme in which naming of values and memory allocation have been separated. Thus there are explicit memory operations (*cons*, *car*, *setcar*, *eq*, etc.) but no assignment to bound variables. The reason for the choice is that it simplifies the semantics and allows one to separate the functional aspects from the imperative ones in a clean way.

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*. Operations are partitioned into algebraic operations and memory operations. The unary memory operations are $\{\mathit{atom}, \mathit{cell}, \mathit{car}, \mathit{cdr}\}$ and binary memory operations are $\{\mathit{eq}, \mathit{cons}, \mathit{setcar}, \mathit{setcdr}\}$. The n-ary algebraic operations are functions mapping $\mathbb{A}^n$ to $\mathbb{A}$. From the given sets we define expressions, value expressions, lambda abstractions, value substitutions, and contexts.

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

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

We let $a, a_0, \ldots$ range over $\mathbb{A}$, $x, x_0, y, z, \ldots$ range over $\mathbb{X}$, $v, v_0, \ldots$ range over $\mathbb{V}$, and $e, e_0, \ldots$ range over $\mathbb{E}$. λ is a binding operator and free and bound variables of expressions are defined as usual. $\mathrm{FV}(e)$ is the set of free variables of e. Two expressions are considered equal if they are the same up to renaming of bound variables. $e\{x := e'\}$ is the result of substituting e' for x in e taking care not to trap free variables of e'. A value substitution is a finite map from variables to value expressions. We let σ, σ_0, …range over value substitutions. e^σ is the result of simultaneously substituting free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$, again taking care not to trap free variables.

Contexts are expressions with holes. We use ε to denote a hole. We let E, E' range over ${}^{\varepsilon}\mathbb{E}$. $E[\![e]\!]$ denotes the result of replacing any holes in E by e. Free variables of e may become bound in this process.

2.2. Operational semantics

An operational semantics based on memory structures and a purely syntactic operational semantics for $\mathbb{E}$ are given in [Mason and Talcott 89c]. We outline the syntactic semantics here, as it provides a natural basis for reasoning about program equivalence.

Computation is a process of stepwise reduction of an expression to a canonical form. In order to define the reduction rules we introduce the notions of *memory context*, *reduction context*, and *primitive expression*. Memory contexts describe memory states and are contexts Γ of the form:

$$\begin{array}{l}\mathtt{let}\{z_1 := cons(\mathtt{Nil},\mathtt{Nil})\}\ldots\mathtt{let}\{z_n := cons(\mathtt{Nil},\mathtt{Nil})\}\\ \qquad\mathtt{seq}(setcar(z_1, v^a_1), setcdr(z_1, v^d_1), \ldots, setcar(z_n, v^a_n), setcdr(z_n, v^d_n), \varepsilon)\end{array}$$

where $z_i \neq z_j$ when $i \neq j$. As descriptions of memories we can view them as finite maps from variables to pairs of value expressions. We define $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$ and $\Gamma(z_i) = [v^a_i, v^d_i]$ for $1 \leq i \leq n$. $\Gamma\{z := [v_a, v_d]\}$ is the memory context Γ' such that $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \cup \{z\}$, $\Gamma'(z) = [v_a, v_d]$, and $\Gamma'(z') = \Gamma(z')$ if $z \neq z'$.

An expression is either a value expression or decomposes uniquely into a primitive expression placed in a reduction context. Reduction contexts identify the subexpression of an expression that is to be evaluated next. The set of reduction contexts, $\mathbb{R}$, is the subset of ${}^{\varepsilon}\mathbb{E}$ defined by

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

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

Primitive expressions describe the primitive computation steps. A primitive step is either the application of a lambda abstraction to a value (beta reduction), branching according to whether a test value is `Nil` or not, or the application of a primitive operation. The set of primitive expressions, $\mathbb{E}_{\mathrm{prim}}$, is defined as

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

Single-step reduction ($\mapsto$) is a relation on pairs $\Gamma; e$ consisting of a memory context and an expression, with $\mathrm{FV}(e) \subseteq \mathrm{Dom}(\Gamma)$. We call such pairs descriptions. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. Single-step reduction is the least relation such that

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

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

(delta) $\quad \Gamma; R[\![\delta(v_1, \ldots, v_n)]\!] \mapsto \Gamma'; R[\![v']\!]$

where in (**delta**) we assume that either δ is an n-ary algebraic operation, $v_1, \ldots, v_n \in \mathbb{A}^n$, $\delta(v_1, \ldots, v_n) = v'$, and $\Gamma = \Gamma'$ or $\Gamma; R[\![\delta(v_1, \ldots, v_n)]\!] \rightharpoondown \Gamma'; R[\![v']\!]$ and, for example,

$$\Gamma; R[\![cell(v)]\!] \rightharpoondown \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } v \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{otherwise} \end{cases}$$

$$\Gamma; R[\![eq(v_0, v_1)]\!] \rightharpoondown \begin{cases} \Gamma; R[\![\mathtt{T}]\!] & \text{if } v_0 = v_1 \text{ and } v_i \in \mathbb{A} \cup \mathrm{Dom}(\Gamma) \text{ for } i < 2 \\ \Gamma; R[\![\mathtt{Nil}]\!] & \text{otherwise} \end{cases}$$

$$\Gamma; R[\![cons(v_0, v_1)]\!] \rightharpoondown \Gamma\{z := [v_0, v_1]\}; R[\![z]\!]$$

$$\Gamma; R[\![car(z)]\!] \rightharpoondown \Gamma; R[\![v_a]\!]$$

$$\Gamma; R[\![setcar(z, v)]\!] \rightharpoondown \Gamma\{z := [v, v_d]\}; R[\![z]\!]$$

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

A value description is a memory context together with a value expression. A description is defined just if it reduces to a value description. When expressions are used where descriptions are expected we take the memory context to be empty.

In order to make programs easier to read we introduce some abbreviations. Multiary application and abstraction is obtained by currying as usual and application is usually represented by juxtaposition rather than explicitly writing out `app`. `let` is lambda-application as usual. `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 also write $null(x)$ for $eq(x, \mathtt{Nil})$.

2.3. Notions of equivalence

Now we define two notions of program equivalence: operational equivalence and constrained equivalence.

Two expressions are operationally equivalent if they cannot be distinguished by any program context. Operational equivalence enjoys many nice properties such as being a congruence relation on expressions. It subsumes the lambda-v-calculus [Plotkin 75] and the lambda-c calculus [Moggi 89]. The theory of operational equivalence for the language used in this paper is presented in [Mason and Talcott 89c].

Constrained equivalence is a relation between sets of constraints (on memory states) and pairs of expressions. The interpretation is that in all contexts satisfying the constraints, evaluation of the expressions is either undefined or produces the same results and has the same effect on memory (modulo garbage).

Constrained equivalence is a stronger relation than operational equivalence and hence is often easier to establish. A version of constrained equivalence for the first-order subset of our language was studied in detail in [Mason 86] and a powerful collection of tools was developed there for reasoning about this relation. An inference system that is complete for zero-order terms (first-order expressions not involving recursively defined functions) is given in [Mason and Talcott 89a,b]. Constrained equivalence restricted to the empty set of constraints implies operational equivalence and is the same as operational equivalence in the first-order case. Constrained equivalence naturally allows reasoning by cases and permits use of a variety of induction principles.

Definition ($\cong$): Two expressions are operationally equivalent, written $e_0 \cong e_1$ just if for any closing context E either both $E[\![e_0]\!]$ and $E[\![e_1]\!]$ are defined or both are undefined.

By definition operational equivalence is a congruence relation on expressions.

Theorem (Congruence): $e_0 \cong e_1 \Rightarrow (\forall E \in {}^{\varepsilon}\mathbb{E})(E[\![e_0]\!] \cong E[\![e_1]\!])$

To define constrained equivalence we need to define notions of predicate, constraint, and satisfaction. A predicate computes a total function with range $\{\mathtt{T}, \mathtt{Nil}\}$ and has no visible effect (i.e. its application leaves existing memory unchanged). For example, *atom* and $\lambda x.\mathtt{and}(\mathit{listp}(x), \mathit{eq}(\mathit{length}(x), n))$ are predicates. A constraint is an expression of one of the following forms: $p(v_1, \ldots, v_n) \simeq \mathtt{T}$, $p(v_1, \ldots, v_n) \simeq \mathtt{Nil}$, $\mathit{car}(v_0) \simeq v_1$, $\mathit{cdr}(v_0) \simeq v_1$, $v_0 \simeq v_1$, and $\neg(v_0 \simeq v_1)$, where p is a predicate. We let φ, φ', … denote constraints and Σ, Σ', … denote finite sets of constraints. To emphasize the logical nature of predicates we abbreviate $p(v_1, \ldots, v_n) \simeq \mathtt{T}$ by $p(v_1, \ldots, v_n)$ and $p(v_1, \ldots, v_n) \simeq \mathtt{Nil}$ by $\neg p(v_1, \ldots, v_n)$.

A pair consisting of a memory context and value substitution $\Gamma; \sigma$ satisfies an equation $e_0 \simeq e_1$ (written $\Gamma, \sigma \models e_0 \simeq e_1$) just if both descriptions $\Gamma; e_j^{\sigma}$ are undefined, or both evaluate to the same value description modulo production of garbage. Similarly we define the notion of a memory context and value substitution satisfying a set of constraints Σ (written $\Gamma; \sigma \models \Sigma$), for details see [Mason and Talcott 89a,b].

Definition (constrained equivalence): Two expressions e_0, e_1 are equivalent under constraints Σ (written $\Sigma \models e_0 \simeq e_1$) just if $\Gamma; \sigma \models \Sigma$ implies $\Gamma; \sigma \models e_0 \simeq e_1$ for any $\Gamma; \sigma$ (subject to simple conditions on free variables).

Since constraint sets define classes of memory contexts we define the notion of domain of a constraint set: $x \in \mathrm{Dom}(\Sigma)$ just if $\Sigma \models \mathit{cell}(x)$. We write $\models e_0 \simeq e_1$ (or just $e_0 \simeq e_1$ when no confusion is possible) for $\emptyset \models e_0 \simeq e_1$.

Theorem (striso): Unconstrained equivalence implies operational equivalence: $e_0 \simeq e_1$ implies $e_0 \cong e_1$.

Recursive functions can be defined using definable fixed-point operators (cf. [Talcott and Mason 89c]). We use the usual recursion equation syntax for such definitions.

Theorem (recdef): If f has been defined by $f(x, y) \leftarrow e$ then we have $f(e_0, e_1) \simeq \mathtt{let}\{x := e_0\}\mathtt{let}\{y := e_1\}e$ (assuming x not free in e_1 which can easily be arranged by renaming).

The following is a sampling of facts regarding the constrained equivalence relation (cf. [Mason and Talcott 89a,b]) (**set.cons**) and (**gc**) allow for the simplification of memory descriptions to canonical form. (**set.set**), (**let.rcx**), (**commutes**) provide mechanisms for rearranging expressions prior to the simplification process mentioned above.

Theorem (Rules of constrained equivalence): Let Φ denote an equation of the form $e_{\mathrm{lhs}} \simeq e_{\mathrm{rhs}}$. Then

(cases) $\Sigma \cup \{p(\bar{v})\} \models \Phi$ and $\Sigma \cup \{\neg(p(\bar{v}))\} \models \Phi$ implies $\Sigma \models \Phi$

(car) $\Sigma \cup \{\mathit{car}(x) \simeq z\} \models \Phi$ implies $\Sigma \models \Phi$ $z \notin \mathrm{FV}(\Phi) \cup \mathrm{FV}(\Sigma)$ and $x \in \mathrm{Dom}(\Sigma)$

(Ri) $\Sigma \models \Phi$ implies $\Sigma \models R[\![\Phi]\!]$

(let.cnv) $\Sigma \models e\{x := v\} \simeq \texttt{let}\{x := v\}e$

(let.rcx) $\Sigma \models R[\![\texttt{let}\{x := e_0\}e_1]\!] \simeq \texttt{let}\{x := e_0\}R[\![e_1]\!] \qquad x \notin \mathrm{FV}(R)$

(let.id) $\Sigma \models e \simeq \texttt{let}\{x := e\}x$

(set.set) $\Sigma \models \texttt{seq}(\mathit{setcdr}(x_0, x_1), \mathit{setcar}(x_2, x_3), e) \simeq \texttt{seq}(\mathit{setcar}(x_2, x_3), \mathit{setcdr}(x_0, x_1), e)$

(set.cons) $\Sigma \models \mathit{setcar}(\mathit{cons}(z, y), x) \simeq \mathit{cons}(x, y)$

Theorem (Garbage collection rule): If Γ is memory context such that $\mathrm{Dom}(\Gamma) \cap \mathrm{FV}(e) = \emptyset$ then $\Gamma[\![e]\!] \simeq e$.

In the presence of effects one is not free to change the order of evaluation of expressions. However expressions that do not 'interfere' with one another can be interchanged. For example two expressions do not interfere if neither one makes use of any *write* operations or if one of the expressions has only *allocate* effect (makes no use of *read* or *write* operations). More precisely we have the following.

Definition (commutes): e_0 and e_1 commute if for all e

$$\texttt{let}\{z_0 := e_0\}\texttt{let}\{z_1 := e_1\}e \simeq \texttt{let}\{z_1 := e_1\}\texttt{let}\{z_0 := e_0\}e$$

provided z_0 not free in e_1 and z_1 not free in e_0.

Definition (effect classes): The sets of operations with read, write, and allocate effect are: $\mathbb{F}_{read} = \{\mathit{car}, \mathit{cdr}\}$, $\mathbb{F}_{write} = \{\mathit{setcar}, \mathit{setcdr}\}$, and $\mathbb{F}_{allocate} = \{\mathit{cons}\}$.

Theorem (commutes): e_0 and e_1 commute if one of the following conditions holds.

(i) e_0, e_1 are built from $\mathbb{F} - \mathbb{F}_{write}$

(ii) e_0 is built from $\mathbb{F} - (\mathbb{F}_{write} \cup \mathbb{F}_{read})$

Although operational equivalence is a congruence relation, constrained equivalence is not preserved by placement into an arbitrary context. Context introduction is only valid if the context does not invalidate the constraints.

Definition (Invalidation): A E does not invalidate Σ if $\Sigma \models e_0 \simeq e_1$ implies $\Sigma \models E[\![e_0]\!] \simeq E[\![e_1]\!]$ for any e_0,e_1.

Theorem (Context introduction): E does not invalidate Σ in the following cases:

(i) Σ is the empty set of constraints and E is any context,

(ii) Σ contains only assertions of the form $\mathit{atom}(x)$, $\neg\mathit{atom}(x)$, $x \simeq y$, or $\neg(x \simeq y)$, and E is any context that does not trap the free variables of Σ, or

(iii) Σ is any constraint set and E is of the form $\texttt{let}\{x := e\}\varepsilon$ where (under constraint Σ) e has no write effect (evaluation of e will not execute any operations in $\mathbb{F}_{\mathrm{write}}$) and x is not free in Σ.

Many Lisp computations are defined by recursion on lists, trees (S-expressions), and other structures. It is natural to reason about such programs using corresponding rules for structural induction. In the presence of effects, one has to be careful in formulating the induction scheme. In particular one must avoid assuming some property of the computation for the *cdr* of a list that is mutated into a longer or cyclic list.

As an example of a correct induction rule we give the rule for List-induction. The predicate $List(x)$ means that $atom(cdr^n(x)) \simeq \mathtt{T}$ for some number n. Let $\mathcal{E}$ be a set of pairs of expressions ($\mathcal{E} \subset \mathbb{E} \times \mathbb{E}$).

Theorem (List-induction): To prove that $\Sigma(x) \cup \{List(x)\} \models e_0 \simeq e_1$ for $(e_0, e_1) \in \mathcal{E}$ it suffices to prove that for each $(e_0, e_1) \in \mathcal{E}$ the following two conditions hold.

(at) $\Sigma(x) \cup \{length(x) \simeq 0\} \models e_0 \simeq e_1$, and

(nonat) for each number n, if for each (e_0', e_1') in $\mathcal{E}$

$$\Sigma(x) \cup \Sigma(x_d) \cup \{length(x) \simeq n+1, x_d \simeq cdr(x)\} \models e_0'\{x := x_d\} \simeq e_1'\{x := x_d\}$$

then $\Sigma(x) \cup \{length(x) \simeq n+1\} \models e_0 \simeq e_1$.

Note that $\{length(x) \simeq 0\} \models null(x)$ and $\{length(x) \simeq n+1\} \models \neg(null(x))$. Induction is treated in more detail in the extended version of [Mason and Talcott 89b].

3. Dreconcing

Our first example is the optimization of the Lisp function *dreconc* (due to Jon L White [private communication], cf. [Gabriel 85], p. 18]). *dreconc* takes two lists, destructively reverses the first and attaches it onto the second. The first program for computing *dreconc* is [using c for current, n for next, and p for previous]

```
$dreconc(c, p) \leftarrow$ prog{$n$}
                 $b$ :cond($null(c)$, return($p$))
                   setq($n$, $cdr(c)$)
                   $rplacd(c, p)$
                   setq($p$, $c$)
                   setq($c$, $n$)
                   go($b$)
```

The optimization is based on observing a three-fold symmery in the use of the program variables. By unrolling the loop twice and doing some `setq` shuffling this program can be transformed into the following equivalent but more efficient version.

```
$dreconc(c,p) \leftarrow$ prog{$n$}
                  $b$ :cond($null(c)$, return($p$))
                    setq($n$, $cdr(c)$)
                    $rplacd(c,p)$
                    cond($null(n)$, return($c$))
                    setq($p$, $cdr(n)$)
                    $rplacd(n,c)$
                    cond($null(p)$, return($n$))
                    setq($c$, $cdr(p)$)
                    $rplacd(p,n)$
                    go($b$)
```

Formalizing the informal argument for the correctness of this transformation is rather messy although it really only involves the assignments to local variables (program structure) and not the effects on list arguments. An alternative approach is to express the algorithm in an equivalent but more functional form (a mechanical transformation), replacing `prog` and `setq` by tail recursion and `let` binding. Thus we define *fdreconc* (*f* for functional)

$$fdreconc(c,p) \leftarrow \mathtt{if}(null(c), p, \mathtt{let}\{n := cdr(c)\}\mathtt{seq}(setcdr(c,p), fdreconc(n,c)))$$

We can now carry out a corresponding optimization: unfold the recursive call and rename the inner bound n to p (which is no longer used); unfold the call again and rename inner bound n to c. This gives the following definition.

```
$fdreconc(c,p) \simeq$ if($null(c), p$,
                    let{$n := cdr(c)$}seq($setcdr(c,p)$,
                         if($null(n), c$,
                             let{$p := cdr(n)$}seq($setcdr(n,c)$,
                                  if($null(p), n$,
                                      let{$c := cdr(p)$}seq($setcdr(p,n)$,
                                           $fdreconc(c,p)$)))))))
```

Verifying the correctness of the latter transformation requires only the use of the simple rules (**recdef**) and (**let.cnv**). This approach can be viewed either as a better programming style (relying on the compiler to produce efficient code), or as the application of mechanical transformation tools to move between the imperative and functional versions.

4. Structural copying and traversal

The fact that a function only depends on the elements of a list argument (and not on the list structure itself) can be expressed by $f(x) \simeq f(\mathit{copy\text{-}list}(x))$ and the fact that f produces new list structure can be expressed by $\mathit{copy\text{-}list}(f(x)) \simeq f(x)$. Similarly for tree structures. Examples are given by the following theorem (function definitions can be found in the appendix).

Theorem (copying):

(ap.copy.x) $\mathit{append}(\mathit{copy\text{-}list}(x), y) \simeq \mathit{append}(x, y)$

(ap.copy.y) $\mathit{append}(x, \mathit{copy\text{-}list}(y)) \simeq \mathit{copy\text{-}list}(\mathit{append}(x, y))$

(fr.copy) $\mathit{fringe}(x) \simeq \mathit{fringe}(\mathit{copy\text{-}tree}(x)) \simeq \mathit{copy\text{-}list}(\mathit{fringe}(x))$

Proof (ap.copy.x): To illustrate our methods we prove by list induction on x that $\mathit{append}(\mathit{copy\text{-}list}(x), y) \simeq \mathit{append}(x, y)$. If $\mathit{length}(x) \simeq 0$ then $\mathit{copy\text{-}list}(x) \simeq x$ and we are done by (**Ri**). Assume $\mathit{length}(x) \simeq n + 1$ and let $x_a \simeq \mathit{car}(x)$, $x_d \simeq \mathit{cdr}(x)$. Then by the definition of *copy-list*

$$\mathit{append}(\mathit{copy\text{-}list}(x), y) \simeq \mathtt{let}\{z_d := \mathit{copy\text{-}list}(x_d)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{append}(z, y).$$

Now by the definition of *append*, rules for *cons*, and (**gc**)

$$\begin{aligned}&\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{append}(z, y)\\ &\quad\simeq \mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{cons}(x_a, \mathit{append}(z_d, y))\\ &\quad\simeq \mathit{cons}(x_a, \mathit{append}(z_d, y))\end{aligned}$$

and consequently

$$\begin{aligned}&\mathtt{let}\{z_d := \mathit{copy\text{-}list}(x_d)\}\mathtt{let}\{z := \mathit{cons}(x_a, z_d)\}\mathit{append}(z, y)\\ &\quad\simeq \mathtt{let}\{z_d := \mathit{copy\text{-}list}(x_d)\}\mathit{cons}(x_a, \mathit{append}(z_d, y)) \qquad \text{by context introduction}\\ &\quad\simeq \mathit{cons}(x_a, \mathit{append}(\mathit{copy\text{-}list}(x_d), y)) \qquad \text{by the } \mathtt{let} \text{ rules}\\ &\quad\simeq \mathit{cons}(x_a, \mathit{append}(x_d, y)) \qquad \text{by the induction hypothesis}\\ &\quad\simeq \mathit{append}(x, y) \qquad \text{by the definition of } \mathit{append}\end{aligned}$$

$\square_{\mathbf{ap.copy.x}}$

Some further examples of what can be expressed via copying are the following.

Theorem (more copying):

(nconc.ap) $\mathit{nconc}(\mathit{copy\text{-}list}(x), y) \simeq \mathit{append}(x, y)$

(map.ap) $\mathit{mapc}(f, \mathit{append}(x, y)) \simeq \mathtt{seq}(\mathit{mapc}(f, \mathit{copy\text{-}list}(x)), \mathit{mapc}(f, y))$

Lists with *copy-list* and S-expressions with *copy-tree* are special cases of structures with copy functions. The idea is that by copying the structure, we can reason about recursion on that structure as if there were no operations with write effect.

Definition (delayable traverse): We say an n-ary operation f traverses structures with recognizers S_i (for example *List*) and copy functions c_i (for example *copy-list*) in a delayable manner if

$$S_1(x_1), \ldots S_n(x_n) \models$$
$$\texttt{let}\{z := f(\bar{x})\}\texttt{seq}(e_0, e_1) \simeq \texttt{let}\{y_i := c_i(x_i)\}_{1 \le i \le n}\texttt{seq}(e_0, \texttt{let}\{z := f(\bar{y})\}e_1)$$

for any e_0, e_1 such that z is not free in e_0 and the y_i are fresh. Note that if f traverses in a delayable manner then $f(x_1, \ldots, x_n) \simeq f(c_1(x_1), \ldots, c_n(x_n))$. A trivial case is where f does not 'touch' an argument and the copy function is the identity function $\mathit{identity} \simeq \lambda x.x$.

Theorem (delaying): The following are examples of delayable traversal: $f = \mathit{cons}$, $c_1 = c_2 = \mathit{identity}$; $f = \mathit{length}$, $c_1 = \mathit{copy\text{-}list}$; $f = \mathit{size}$, $c_1 = \mathit{copy\text{-}tree}$; $f = \mathit{append}$, $c_1 = \mathit{copy\text{-}list}$, $c_2 = \mathit{identity}$; and $f = \mathit{fringe}$, $c_1 = \mathit{copy\text{-}tree}$.

Delaying is an additional tool for rearranging expressions. It is used in combination with the copy theorems to eliminate redundant copies. An application of the delaying tecnique is in the proof of the following theorem.

Theorem (tr.map.fr): $\mathit{traverse}(f, \mathit{copy\text{-}tree}(x))$ and $\mathit{mapc}(f, \mathit{fringe}(x))$ have the same effect. I.e. for any e

$$\texttt{seq}(\mathit{traverse}(f, \mathit{copy\text{-}tree}(x)), e) \simeq \texttt{seq}(\mathit{mapc}(f, \mathit{fringe}(x)), e)$$

where *traverse* traverses a tree structure applying its first argument to each leaf.

Proof (tr.map.fr): We show by S-expression induction that

$$\texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{traverse}(f, y), e)$$
$$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{mapc}(f, \mathit{fringe}(y)), e)$$

for any e', e with y not free. (Note that if x is not an S-expression then $\mathit{copy\text{-}tree}(x)$ is undefined and we are done.)

($\mathit{atom}(x)$)

$$\texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{traverse}(f, y), e)$$
$$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', f(y), e)$$
$$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{mapc}(f, \mathit{fringe}(y)), e)$$

($\neg\mathit{atom}(x)$)

$$\texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{traverse}(f, y), e)$$

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{let}\{y_a := \mathit{car}(y)\}\texttt{let}\{y_d := \mathit{cdr}(y)\}$
$\quad \texttt{seq}(e', \mathit{traverse}(f, y_a), \mathit{traverse}(f, y_d), e)$

unfolding and context introduction

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{let}\{y_a := \mathit{car}(y)\}\texttt{let}\{y_d := \mathit{cdr}(y)\}$
$\quad \texttt{seq}(e', \mathit{mapc}(f, \mathit{fringe}(y_a)), \mathit{mapc}(f, \mathit{fringe}(y_d)), e)$

using the induction hypothesis twice

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{let}\{y_a := \mathit{car}(y)\}\texttt{let}\{y_d := \mathit{cdr}(y)\}$
$\quad \texttt{let}\{z_a := \mathit{fringe}(y_a)\}\texttt{let}\{z_d := \mathit{fringe}(y_d)\}$
$\quad\quad \texttt{seq}(e', \mathit{mapc}(f, z_a), \mathit{mapc}(f, z_d), e)$

by (**delaying**) for *fringe*

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{let}\{y_a := \mathit{car}(y)\}\texttt{let}\{y_d := \mathit{cdr}(y)\}$
$\quad \texttt{let}\{z_a := \mathit{fringe}(y_a)\}\texttt{let}\{z_d := \mathit{fringe}(y_d)\}$
$\quad\quad \texttt{seq}(e', \mathit{mapc}(f, \mathit{append}(z_a, z_d)), e)$

by (**map.app**) and (**fr.copy**)

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{let}\{y_a := \mathit{car}(y)\}\texttt{let}\{y_d := \mathit{cdr}(y)\}$
$\quad \texttt{seq}(e', \mathit{mapc}(f, \mathit{fringe}(\mathit{cons}(y_a, y_d))), e)$

by (**delaying**) for *fringe*

$\simeq \texttt{let}\{y := \mathit{copy\text{-}tree}(x)\}\texttt{seq}(e', \mathit{mapc}(f, \mathit{fringe}(y)), e)$

by (**delaying**) for *cons*

$\square$**tr.map.fr**

5. Abstract objects

Abstract objects exhibit the non-inheritance aspects of object-oriented programming. An abstract object is a function with local store. Abstract objects provide a means of encapsulating features of a structure and controlling access to that structure. The idea is that the local store can only be changed by sending a message to the object. The operations on the encapsulated structure are determined by the messages accepted by the object.

We illustrate these ideas for the special case of accumulators. An accumulator object accumulates a sequence of the things sent to it (via a $\texttt{<put,}x\texttt{>}$ message) and responds to a `<get>` message by returning the sequence collected. If $\mathit{mkac}(y)$ creates an accumulator object with initial contents the elements of y, then it mus satisfy the following three laws:

Specification (Accumulator behavior):

(put) $\quad \texttt{let}\{a := \mathit{mkac}(y)\}\texttt{seq}(a(\texttt{<put,}x\texttt{>}), e)$

$$\simeq \texttt{let}\{a := \mathit{mkac}(\mathit{append}(y, \mathit{cons}(x, \texttt{Nil})))\}e$$

(get) $\texttt{let}\{a := \mathit{mkac}(y)\}\texttt{let}\{z := a(\texttt{<get>})\}e$

$$\simeq \texttt{let}\{a := \mathit{mkac}(y)\}\texttt{let}\{z := \mathit{copy\text{-}list}(y)\}e$$

(delay) $\texttt{let}\{a := \mathit{mkac}(y)\}\texttt{seq}(e_0, e_1)$

$$\simeq \texttt{let}\{y' := \mathit{copy\text{-}list}(y)\}\texttt{seq}(e_0, \texttt{let}\{a := \mathit{mkac}(y')\}e_1)$$

if a not free in e_0 and y' fresh

A property that can be proved using only the above accumulator laws will hold for any constructor that satisfies these laws. For example, from the accumulator laws we can prove that an accumulator can be cloned and that traversing accumulates the fringe of an S-expression.

Definition (traversing with an object):

$$\mathit{tro}(f, x) \leftarrow \texttt{seq}(\mathit{traverse}(\lambda y.f(\texttt{<put}, y\texttt{>}), x), f)$$

Theorem (accum): If *mkac* makes accumulator objects and x is an S-expression then

(clone) $\mathit{mkac}(y) \simeq \texttt{let}\{a := \mathit{mkac}(y)\}\mathit{mkac}(a(\texttt{<get>}))$

(tr) $\texttt{let}\{a := \mathit{mkac}(y)\}\mathit{tro}(a, x) \simeq \mathit{mkac}(\mathit{append}(y, \mathit{fringe}(x)))$

There are many possibilities for constructing accumulator objects. One example is the following.

Definition (accumulator object constructor):

$$\mathit{mkac}(x) \simeq \texttt{let}\{z := \mathit{cons}(\texttt{Nil}, \mathit{copy\text{-}list}(x))\}\texttt{seq}(\mathit{setcar}(z, \mathit{last}(z)), \mathit{acob}(z))$$

$$\mathit{acob}(z)(\texttt{<get>}) \simeq \mathit{copy\text{-}list}(\mathit{cdr}(z))$$

$$\mathit{acob}(z)(\texttt{<put}, x\texttt{>}) \simeq \texttt{seq}(\mathit{setcdr}(\mathit{car}(z), \mathit{cons}(x, \texttt{Nil})), \mathit{setcar}(z, \mathit{cdr}(\mathit{car}(z))), \texttt{Nil})$$

$\mathit{acob}(z)(\Gamma) \simeq \texttt{Nil}$ for any other message

Theorem (accum.mk): *mkac* as defined above satisfies the accumulator behavior properties.

An alternative method for reasoning about abstract objects is to reason about their corresponding behavior functions. Behavior functions are a generalization of streams. In response to a message, a pair is returned consisting of a new behavior function (representing the updated local store) together with the reply. The function *bh2ob* makes an object out of a behavior function. It stores the behavior function representing the current local store in its memory.

$$\mathit{bh2ob}(f) \leftarrow \mathit{bhob}(\mathit{cons}(f, \texttt{Nil}))$$

$$bhob(z)(m) \leftarrow \texttt{let}\{y := car(z)(m)\}\texttt{let}\{f := car(y)\}\texttt{let}\{r := cdr(y)\} \\ \texttt{seq}(setcar(z, f), r)$$

We can take behavior functions as specifications of classes of objects, defining an object to be in the specified class just if it is equivalent to $bh2ob(f)$ for some f in the class.

As an example, if we replace traversal with an object (*tro*) by traversal with a stream (*trs*) we can reason about the effect of traversal in terms of effect-free streams and transfer the results to objects using the behavior transfer theorem.

Definition (traversing with a stream):

$$\begin{aligned} trs(s, x) \leftarrow \texttt{if}(&atom(x), \\ &\texttt{let}\{y := s(\texttt{<put}, x\texttt{>})\}car(y), \\ &\texttt{let}\{s := trs(s, car(x))\}trs(s, cdr(x))) \end{aligned}$$

Theorem (behavior transfer): For S-expressions x

$$\texttt{let}\{f := bh2ob(s)\}tro(f, x) \simeq \texttt{let}\{s := trs(s, x)\}bh2ob(s)$$

To continue the accumulator example, we define $mkacstr(y)$ to be a constructor of accumulator behavior functions.

$$mkacstr(y) \leftarrow acstr(copy\text{-}list(y))$$

$$acstr(y)(\texttt{<get>}) \leftarrow cons(acstr(y), copy\text{-}list(y))$$

$$acstr(y)(\texttt{<put}, x\texttt{>}) \leftarrow cons(acstr(append(y, cons(x, \texttt{Nil}))), \texttt{Nil})$$

$$acstr(y)(\Gamma) \leftarrow cons(acstr(y), \texttt{Nil})$$

Now we can show (by an easy S-expression induction using effect-free reasoning) that $trs(mkacstr(y), x) \simeq mkacstr(append(y, fringe(x)))$. Using this and the transfer theorem gives an alternative proof of (**accum.tr**).

6. Discussion

In this paper we have presented techniques for reasoning about programs in a Lisp- (Scheme-, ML-) like language with higher-order functions and objects with memory. The techniques were based on equational theories developed for this language. Although much work remains to develop informal reasoning methods that can readily be expanded to formal proofs, we feel that much progress has been made. Further applications of these methods can be found in the extended versions of [Mason and Talcott 89b,c]. In particular, we have developed a method called simulation induction for proving equivalence of abstract objects.

Another important language feature is the ability to express control abstractions such as escape mechanisms, and coroutining. [Felleisen and Hieb 90] defines reduction calculi extending the call-by-value lambda calculus to languages with control and assignment abstractions. This provides an important basis for axiomatizing program equivalence, however the calculus by its very nature is too weak to support the kind of reasoning we illustrated above. [Talcott 89] axiomatizes program equivalence for a language with higher-order functions and control abstractions (no objects with memory) within a full first-order theory of operations and classes. This provides a very rich theory for expressing and proving properties of programs and many examples including behavior functions, streams, escape mechanisms, and coroutines are worked out in detail.

We have used the notions of effect and interference informally to give intuitive explanations of technical properties. These notions are not new. [Reynolds 89] 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 is 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 88] 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. For example in [Wadler 90] 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 90]. 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.

7. References

Felleisen, M. and Hieb, R. [1989] The Revised Report on the Syntactic Theories of Sequential Control and State, Department of Computer Science, Rice University Technical Report Rice COMP TR89-100.

Gabriel, R. P. [1985] Performance and Evaluation of Lisp Systems, (MIT Press).

Guzmân, J. C. and Hudak. P [1990] Single-Threaded Polymorphic Lambda Calculus. *Fifth annual symposium on logic in computer science*, (IEEE).

Lucasen, J. M. and Gifford, D. K. [1988] Polymorphic effect systems, in: 16th annual ACM symposium on principles of programming languages, pp. 47–57.

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

Mason, I. A. and Talcott, C. L. [1989a] A Sound and Complete Axiomatization of Operational Equivalence between Programs with Memory, Stanford University

Computer Science Department Report STAN–CS–89–1250. (Revised and extended version to appear in *Theoretical Computer Science*).

Mason, I. A. and Talcott, C. L. [1989b] Axiomatizing Operational Equivalence in the presence of Side Effects, in: *4th Symposium on logic in computer science, Asilomar CA*, (IEEE).

Mason, I. A. and Talcott, C. L. [1989c] Programming, Transforming, and Proving with function abstractions and memories. in: *Proceedings of the 16th EATCS Colloquium on Automata, Languages and Programming, Stresa Italy*, Lecture notes in computer science, **372**, (Springer-Verlag). (Revised and extended version submitted for publication.)

Moggi, E. [1989] Computational lambda-calculus and monads, *Fourth annual symposium on logic in computer science*, (IEEE).

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

Reynolds, J. C. [1989] Syntactic Control of Interference, II in: *Proceedings of the 16th EATCS Colloquium on Automata, Languages and Programming, Stresa Italy*, Lecture notes in computer science, **372**, (Springer-Verlag)

Talcott, C. [1989] Programming and proving with function and control abstractions, Stanford University Computer Science Department Technical report STAN-CS-89-1288.

Wadler, P. [1990] Linear types can change the world! *IFIP working conference on programming concepts and methods.* Sea of Gallilee, Israel.

8. Appendix: function definitions

$$\mathit{copy\text{-}tree}(x) \leftarrow \mathtt{if}(\mathit{atom}(x), x, \mathit{cons}(\mathit{copy\text{-}tree}(\mathit{car}(x)), \mathit{copy\text{-}tree}(\mathit{cdr}(x))))$$

$$\mathit{copy\text{-}list}(x) \leftarrow \mathtt{if}(\mathit{atom}(x), x, \mathit{cons}(\mathit{car}(x), \mathit{copy\text{-}list}(\mathit{cdr}(x))))$$

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

$$\mathit{fringe}(x) \leftarrow \mathtt{if}(\mathit{atom}(x), \mathit{cons}(x, \mathtt{Nil}), \mathit{append}(\mathit{fringe}(\mathit{car}(x)), \mathit{fringe}(\mathit{cdr}(x))))$$

$$\mathit{nconc}(x, y) \leftarrow \mathtt{if}(\mathit{atom}(x), y, \mathtt{seq}(\mathit{nc}(x, y), x))$$

$$\mathit{nc}(x, y) \leftarrow \mathtt{if}(\mathit{atom}(\mathit{cdr}(x)), \mathit{setcdr}(x, y), \mathit{nc}(\mathit{cdr}(x), y))$$

$$\mathit{mapc}(f, x) \leftarrow \mathtt{if}(\mathit{atom}(x), \mathtt{Nil}, \mathtt{seq}(f(\mathit{car}(x)), \mathit{mapc}(f, \mathit{cdr}(x))))$$

$$\mathit{traverse}(f, x) \leftarrow \mathtt{if}(\mathit{atom}(x), f(x), \mathtt{seq}(\mathit{traverse}(f, \mathit{car}(x)), \mathit{traverse}(f, \mathit{cdr}(x))))$$

$$\mathit{tro}(f, x) \leftarrow \mathtt{seq}(\mathit{traverse}(\lambda y.f(\mathtt{<put}, y\mathtt{>}), x), f)$$