M-LISP: A Representation-Independent Dialect of LISP with Reduction Semantics[*]

Robert Muller[†]
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138
muller@harvard.edu

Abstract

In this paper we introduce M-LISP, a simple new dialect of LISP which is designed with an eye toward reconciling LISP's metalinguistic power with the *structural* style of operational semantics advocated by Plotkin [Plo75]. We begin by reviewing the original definition of LISP [McC61] in an attempt to clarify the source of its metalinguistic power. We find that it arises from a problematic clause in this definition. We then define the abstract syntax and operational semantics of M-LISP, essentially a hybrid of M-expression LISP and Scheme. Next, we tie the operational semantics to the corresponding equational logic. As usual, provable equality in the logic implies operational equality.

Having established this framework we then extend M-LISP with the metalinguistic *eval* and *reify* operators (the latter is a non-strict operator which converts its argument to its metalanguage representation.) These operators *encapsulate* the metalinguistic representation conversions that occur globally in S-expression LISP. We show that the naive versions of these operators render LISP's equational logic inconsistent. On the positive side, we show that a naturally restricted form of the *eval* operator is confluent and therefore is a conservative extension of M-LISP. Unfortunately, we must weaken the logic considerably to obtain a consistent theory of reification.

1 Introduction

Consider an interpretive programming environment for some applicative language $\mathcal{L}$. Let us say that the $\mathcal{L}$-program P is a *metaprogram* of some kind — it operates on an $\mathcal{L}$-representation $\bar{p}$ of some $\mathcal{L}'$-program p. As a typical example, P may be a *compiler* from source language $\mathcal{L}'$ to object language $\mathcal{L}$. When applied to an $\mathcal{L}$-representation $\bar{p}$, of a source $\mathcal{L}'$-program p it reduces to an $\mathcal{L}$-representation, $\bar{o}$, of the $\mathcal{L}$-object program o.

$$P\ \bar{p} \twoheadrightarrow \bar{o}$$

[*]This is a revised and expanded version of [Mul91]. The author's current email address is `muller@cs.bc.edu`.

[†]This research was supported in part by Defense Advanced Research Projects Agency N00039-88-C-0163.

With the representation $\bar{o}$ in hand then, how might the object program o be executed? One method is to have another $\mathcal{L}$-program I, a universal program which interprets $\bar{o}$ according to the semantics of $\mathcal{L}$. While the object program can be faithfully interpreted in this way, in practice it is too inefficient since it introduces a second layer of evaluation.

In LISP this inefficiency can be avoided. An explicit application of its *eval* operator entails a *dynamic decoding* from $\bar{o}$ to o and, given that o represents a LISP program, the object program can then be executed directly by the same evaluator which executed P. The decoding performed by the LISP virtual machine is an essential feature of LISP — it mimics the implicit decoding that takes place on the underlying *stored-program machine* in which program o and data $\bar{o}$ are represented by the very same bits in memory.

In this paper we set out to reconcile this metalinguistic power with the structural style of operational semantics that has been persuasively advocated by Plotkin [Plo75, Plo81]. As he recounts in [Plo81], Plotkin was first motivated to introduce this style by the λ-calculus and its fundamental notion of β-reduction. He first applied the idea to the analysis of ISWIM, a language with roots in λ-calculus and M-expression LISP. In recent years the structural approach has been used to describe a wide variety of programming language features.

LISP's metalinguistic power, on the other hand, seems tied to a longstanding practice of programming in its symbolic representation language and to the anomalous operational behavior of its quotation form. For example, any reduction semantics for LISP must relate the application

$$\texttt{(car (quote (a . b)))} \tag{1}$$

to its value, the symbol:

$$\texttt{a} \tag{2}$$

It must also relate the term

$$\texttt{(cons (quote car) (quote ((quote (a . b)))))} \tag{3}$$

to its value, the list structure:

$$\texttt{(car (quote (a . b)))} \tag{4}$$

Since the list in (4) is identical to the application in (1) the operational rules must be defined in such a way as to distinguish them — otherwise (3) would be erroneously related to (2)[1]. Typically there is a "reduction" rule for quoted pairs which recursively propagates quotation inward to the atoms. For example, the rule

$$(\text{quote } (s_1 \text{ . } s_2)) \to (\text{cons } (\text{quote } s_1) \ (\text{quote } s_2)) \tag{5}$$

would reduce (3) to

```
(cons (quote car)
      (cons (cons (quote quote)
            (cons (cons (quote a) (quote b)) (quote nil))) (quote nil)))
```

[1]The relation we are referring to here is the reflexive and transitive closure $\twoheadrightarrow_R$ of the single step reduction relation $\longrightarrow_R$. In the induced evaluator (and in the corresponding *natural semantics* [Kah98]) the problem manifests itself as lack of *idempotence*.

But then, having distinguished values from redexes, one is hard-pressed to explain the metalinguistic facilities. The kind of behavior illustrated in (1) through (4) is generally understood to be essential. For example, an explicit application of the *eval* operator to (3)

```
(eval (cons (quote car) (quote ((quote (a . b))))))          (6)
```

should be related to (2). It is also easy to see the disabling effect of this distinguishing reduction rule on LISP's indispensable *expression macros*: macro expansion functions generate representations of expressions rather than expressions. Moreover, this problem remains when the lists are constructed via the *cons* or *list* operators.

It turns out that the anomalous behavior illustrated above and the accompanying metalinguistic power arise from a simple error in the original definition of LISP [McC61] — the definition of *quote* in the function which codes M-expressions as S-expressions for *eval* is incorrect. If the coding function is corrected and the appropriate changes to *eval* are made then the resulting language has a more reasonable operational semantics but it has no metalinguistic power whatsoever.

So it seems natural to avoid the problematic coding algorithm altogether and consider how the metalinguistic facilities might be reintroduced directly in (a Scheme-style variant of) the uncoded language. We call this hybrid of M-expression LISP and Scheme, M-LISP. Since this dialect is independent of McCarthy's original representation function it has no *quote* form or any of its related forms *backquote*, *unquote* or *unquote-splicing*. Its essential abstract syntax is

$$M ::= X \mid [] \mid [M \,.\, M] \mid x \mid (\lambda x.M) \mid (M\ M) \mid (\texttt{IF}\ M\ M\ M)$$

where X denotes the set of upper-case symbols, LISP's symbolic constants, $[M \,.\, M]$ denotes a pair and x denotes the set of lower-case symbols, LISP's variables. The M-LISP terms corresponding to (1) through (4) are `(car [A . B])`, `A`, `[CAR . [[QUOTE [A . B]]]]` and `[CAR [QUOTE [A . B]]]` respectively. The question that remains to be answered then is how LISP's various metalinguistic facilities are to be reintroduced in the new dialect. In this paper we consider the *eval* and *reify* operators, the latter is a non-strict operator which converts its argument to its representation[2]. We consider syntax macros elsewhere [Mul90].

Although our main objective in this paper is to come to grips with the operational aspects of these metalinguistic facilities, we believe that it is also important to clear up the longstanding confusion about the source of LISP's metalinguistic power — contrary to folklore, it is unrelated to the fact that LISP programs are represented by lists. To make this clear, in Section 2 we review enough of [McC61] to illustrate the troublesome definition cited above. This will also help clarify how the metalinguistic facilities might be integrated in the new dialect.

After this review we set out to study the problem within the framework laid out by Plotkin in [Plo75]. We define a deterministic structured operational semantics for the pure subset of M-LISP. We then establish the connection to the corresponding equational logic in the usual way and prove its consistency. Finally we establish the correspondence of the

[2]It would be natural to call these operators *unquote* and *quote*, respectively. Unfortunately these names are already used in S-expression LISP.

logic to the observational congruence relation, viz., terms that are provably equal behave the same way in all program contexts.

Having established that the pure subset is a reasonable language we then define the axioms and inference rules which give a deterministic operational semantics for the *eval* and *reify* operators. While the new operators achieve the same ends as their S-expression LISP counterparts, they do so in somewhat different ways. In particular, the new versions encapsulate within the operators the representation conversions which occur globally in S-expression LISP. For example, M-LISP's analog of (6) is

```
(eval [APP [IDENT CAR] [PAIR [SYMBOL A] [SYMBOL B]]])          (7)
```

The argument to *eval* reduces to a standard representation of a term. The *eval* operator then simply decodes it (in this case yielding the term (car `[A . B]`)) and the result is then reduced to a value (in this case `A`). A complimentary idea holds for the *reify* operator. For example,

(reify (car `[A . B]`))

evaluates to the standard representation of its argument

```
[APP [IDENT CAR] [PAIR [SYMBOL A] [SYMBOL B]]].
```

Turning to the equational logics, we confirm the folklore that these constructions are in some sense unreasonable: the nondeterministic notion of reduction obtained by taking the union of β_{π_v} and an *eval* operator corresponding to S-expression LISP's is not Church-Rosser (CR). Similarly $reify \cup \beta_{\pi_v}$ is not CR. It is then straightforward to show that the corresponding equational logics are inconsistent. On the positive side we show that a naturally restricted form of the *eval* operator is CR and is therefore a conservative extension of M-LISP. But we are forced to weaken the equational theory to obtain a consistent theory for the *reify* operator.

The work described here was undertaken in response to our misgivings about the above described anomalies. (We have found that they are widely shared.) In particular, we were unfamiliar with any satisfactory explanation of the source of LISP's metalinguistic power or of its connection to the anomalies. We set out to provide one. In identifying the problem in [McC61], we believe that we have, in effect, stumbled across the essential pure LISP: call-by-value (dynamically typed) polymorphic λ-calculus extended with five constant functions (with all other constants serving as uninterpreted symbols), a syntactic constructor for pairs and a conditional construction. In our view, this development is the primary contribution of this work.

The metalinguistic operators considered in the sequel can be taken as yet another in a long tradition of studies of extensions of pure LISP. Our main purpose in introducing these extensions is to show how this might be accomplished in an otherwise representation-independent language. Although it is not our main purpose here to justify or advocate these operators as language features, a strong case can be made for the *eval* operator. It incorporates in the LISP interpreter a crucial property of the underlying stored-program machine model: the ability for programs to dynamically construct programs which can subsequently be executed by the processor (i.e., without an embedded evaluator.) For

$$\begin{array}{lcll}
f & ::= & x \mid \lambda[[x;...;x];e] \mid \text{label}[x;f] & \text{— S-functions} \\
e & ::= & s \mid x \mid f[e;...;e] \mid [e \rightarrow e, e] & \text{— Expressions} \\
s & ::= & X \mid () \mid (s \, . \, s) & \text{— S-expressions}
\end{array}$$

Figure 1: The Abstract Syntax of M-expression LISP.

example, a *linker* operates on some suitable (i.e., normal-form) *representation* $\bar{P}$ of its argument program P. The linker performs its task on its data $\bar{P}$, and the representation $\bar{P}'$ of the linked program is stored in memory. But the processor can then execute P' *directly* as a program. The act of loading the program counter with the beginning address of $\bar{P}'$ effectively *decodes* $\bar{P}'$ to P' in much the same way that the explicit *eval* operator decodes representations of LISP programs. We are unaware of any formal treatment of this property of systems programming languages.

We do not believe, on the other hand, that a persuasive case for the practical importance of the *reify* operator has been made — it appears that many of its applications in LISP can be carried out with macros [Pit80]. Smith [Smi84] and the researchers that followed make the case that reifiers allow a programmer to install procedures in the implementation. We include reification in our study in part because of its symmetry with *eval* and in an effort to present a complete picture of how LISP's metalinguistic features can be reintroduced.

Finally, to avoid any confusion we wish to stress that our concerns in this paper are not with LISP's notation but rather with the evaluator's dynamic representation conversions and the points in the evaluation process at which these conversions occur. We believe that as a matter of language design, it is better to encapsulate the conversions within the operators that require them rather than to have the decoding occur universally as it does in S-expression LISP dialects such as Scheme.

The remainder of this paper is organized as follows. In Section 2 we motivate M-LISP by reviewing enough of [McC61] to illustrate the problematic definition. This extended account of the source of LISP's metalinguistic power can safely be skipped by those who already have the idea. In Section 3 we develop the operational semantics and equational logic of pure M-LISP. Section 4 develops the semantics of the extended language. In Section 4.1 we define criteria for adequate representations of terms. In Section 4.2 we define the *eval* operator and in Section 4.3 we define the *reify* operator. In each case we consider their effect on the logic. In Section 5 we compare the present work to Smith's *reflective* 3-LISP and other related work. In Section 6 we draw some conclusions.

2 Background

Recall that LISP was introduced as a formalism for coding recursive functions over symbolic expressions (S-expressions). The abstract syntax of the meta-expressions (M-expressions) is presented in Figure 1. We informally describe the syntax and the semantics of M-expression LISP. The symbol x denotes the lexical category of lower-case symbols. Its leftmost occurrence in the first production defines the *function variables* which range over the McCarthy primitives $\{car, cdr, cons, eq?, atom?\}$ as well as functions defined with λ and *label*. The clause $\lambda[[x;...;x];e]$ defines the form of *call-by-value procedures*. This notation was inspired

```
apply[fn;x;a] =
  [atom[fn] --> [eq[fn;CAR] --> caar[x];
                 eq[fn;CDR] --> cdar[x];
                 eq[fn;CONS] --> cons[car[x];cadr[x]];
                 eq[fn;ATOM] --> atom[car[x]];
                 eq[fn;EQ] --> eq[car[x];cadr[x]];
                 T --> apply[eval[fn;a];x;a]];                      (8)
   eq[car[fn];LAMBDA] --> eval[caddr[fn];pairlis[cadr[fn];x;a]];    (9)
   eq[car[fn];LABEL] --> apply[caddr[fn];x;
                               cons[cons[cadr[fn];caddr[fn]];a]]

eval[e;a] = [atom[e] --> cdr[assoc[e;a]];
      atom[car[e]] -->
               [eq[car[e];QUOTE] --> cadr[e];                      (10)
                eq[car[e];COND] --> evcon[cdr[e];a];
                T --> apply[car[e];evlis[cdr[e];a];a]];
      T --> apply[car[e];evlis[cdr[e];a];a]]
```

Figure 2: The Universal Functions.

by Church's notation for λ-abstractions. The occurrences of x in the formal parameter positions are *value variables* which range over S-expressions. The label expression can be used in conjunction with the *conditional* $[e \to e, e]$, to define recursive functions — one of LISP's key innovations.

Turning to the second production, the s category denotes the syntactic category of S-expression constants. The other expression clauses consist of value variables x, procedure *applications* $f[e;...;e]$, and the conditional $[e \to e, e]$. Since LISP is call-by-value, the arguments in an application $f[e;...;e]$ are evaluated before the procedure is applied. Finally, the symbol X denotes the lexical category of upper-case symbols; LISP's *symbolic constants.* Note that the constants are distinguished from variables by case. Proper lists can be abbreviated in the concrete syntax by the following dot elimination rule

$$(M_1\ M_2\ \dots\ M_n) \equiv (M_1\ .\ (M_2\ .\ (\ \dots\ (M_n\ .\ ())\ \dots\ ))).$$

2.1 The Expressiveness of LISP

In order to demonstrate the expressiveness of the formalism McCarthy defined a universal S-function *apply* which given a representation $\mathcal{R}(f)$ of an arbitrary S-function f could simulate its application to arbitrary S-expression arguments $s_1\ \dots\ s_n$ such that

$$\mathit{apply}[\mathcal{R}(f); (s_1\ \dots\ s_n); ()] = f[s_1;\ \dots\ ; s_n], \tag{11}$$

when either side is defined. The universal functions *apply* and *eval* are reproduced from [McC62] pp. 13 in Figure 2. There are some subtle differences between this evaluator and

$$
\begin{aligned}
\mathcal{R}(x) &\equiv X \\
\mathcal{R}(\lambda[[x_1;...;x_n];e]) &\equiv (\texttt{LAMBDA}\ (\mathcal{R}(x_1)\ ...\ \mathcal{R}(x_n))\ \mathcal{R}(e)) \\
\mathcal{R}(\text{label}[x;f]) &\equiv (\texttt{LABEL}\ \mathcal{R}(x)\ \mathcal{R}(f)) \\
\mathcal{R}(s) &\equiv (\texttt{QUOTE}\ s) \qquad (12) \\
\mathcal{R}(f[e_1;...;e_n]) &\equiv (\mathcal{R}(f)\ \mathcal{R}(e_1)\ ...\ \mathcal{R}(e_n)) \\
\mathcal{R}([e_1 \to e_2, e_3]) &\equiv (\texttt{IF}\ \mathcal{R}(e_1)\ \mathcal{R}(e_2)\ \mathcal{R}(e_3))
\end{aligned}
$$

Figure 3: The Representation of M-expressions.

f	$::=$	x | (lambda (x ... x) e) | (label x f)	— Functions
e	$::=$	(quote s) | x | (f e ... e) | (if e e e)	— Expressions
s	$::=$	x | () | (s . s)	— S-expressions

Figure 4: The Abstract Syntax of S-expression LISP.

that defined in [McC61] but they have no bearing on the issue in question here.

The representation map $\mathcal{R}$: *M-expression* $\to$ *S-expression* defined for this recursion-theoretic argument is given by induction on the structure of M-expressions in Figure 3. We will freely use lowercase symbols for the S-expression image of $\mathcal{R}$ when no confusion arises. The abstract syntax of the resulting representation language, which we will refer to as S-expression LISP, is shown in Figure 4.

2.2 Assessing Apply

On purely recursion-theoretic grounds *apply* is satisfactory — it can be shown that equation (11) is satisfied and therefore the language is universal in this sense. The condition effectively characterizes the extensional class of partial functions over S-expression that are LISP-definable. On the other hand, it is well-known that circumstances changed this recursion-theoretic exercise to one in programming language semantics. The evaluator is then taken as a definitional interpreter (cf., [Rey72]) specifying the computational *behavior* of LISP. So we are now concerned with a host of qualitative computational issues such as evaluation order and scoping discipline.

It is well-known that the evaluator is not completely satisfactory from this perspective. The evaluator's iterated evaluation of function variables, clause (8), and *dynamic scoping*, see clause (9), are two of its widely studied properties. The former behavior is intended to look up the value of recursive function variables defined via *label*. But it can also be used to reach through chains of *variable/symbol* bindings to arrive at function values. For example, in LISP 1.5, the expression:

```
((lambda (x) ((lambda (f) (f (quote (a . b)))) (quote x))) (quote car))
```

evaluates to `a`. The dynamic scoping property can lead to the unintended capture of free variables when functions are passed as arguments or returned as results (i.e., the *funarg* problem.)

We are interested in a related but more fundamental aspect of the evaluator: its metalinguistic power. The key to this power is the non-inductive coding of the inductively defined S-expression constants, equation (12). Note that this defines a base case for the representation language despite the fact that s does not define a base case for the represented language (i.e., S-expressions have structure). As a consequence, the representation of a pair $(s_1\ .\ s_2)$ is not a composition of the representations of its components s_1 and s_2. Thus, when a projection function is applied the component term is returned *uncoded.* This all but forces the evaluator's *decoding* clause, (10), as well as the uncoded form of pairs constructed by the clause for the *cons* operator and the tagless references in the *atom* and *eq* clauses. The LISP evaluator operates on uncoded (or decoded) expression representations.

In most cases no harm is done in ignoring the distinction between an object and its representation. Such *use/mention* errors are common and often useful. In this context, however, it is important to recall that evaluators only mention the objects of interest in computation (i.e., expressions) by using their representations. The combination of clauses (12) and (10) effects a shift between the mention and the use of (*cadr e*). When (*cadr e*) is itself the representation of some program phrase then metalinguistic power results.

2.3 Coding Constants Inductively

Let us consider the effect of replacing equation (12) by a well-defined structural induction, say:

$$\begin{array}{rcl} \mathcal{R}(X) & \equiv & (\texttt{SYMBOL}\ X) \\ \mathcal{R}(()) & \equiv & (\texttt{NIL}) \\ \mathcal{R}((s_1\ .\ s_2)) & \equiv & (\texttt{PAIR}\ \mathcal{R}(s_1)\ \mathcal{R}(s_2)), \end{array}$$

and making the corresponding changes in *eval* and *apply*, Figure 5. First, the inductive coding has no effect on the extensional class of functions which are LISP-definable and therefore the change has no effect on the original recursion-theoretic claim. Second, note that the distinguishing rule exhibited in (5) is effectively coding constant subterms which (12) has failed to code. This rule is no longer required[3]. As one would expect, there is a natural correspondence between the semantics of the inductively defined representation language and that of the representation-independent language M-expression LISP: given an S-expression representation $\bar{e}$ of an M-expression $e = s$, the evaluator produces the S-expression representation $\bar{s}$ of the answer s. Moreover, the evaluation model can be described in a simple reduction setting along the lines of the usual applicative languages.

On the other hand, the inductive representation leads to a severe restriction for S-expression LISP: dynamic representation decoding is essential to the expression of higher-order functions. It effectively side-steps the restriction in the abstract syntax of M-expressions that precludes functional arguments. For example, although $mapcar[\lambda[[x];x];lst]$ is not a syntactically well-formed M-expression, the same effect is achieved via the decoding by

[3]We are pleased to dispense with it since such rules ordinarily specify run-time computation steps and one would not expect that they should be applied to constants.

```
apply[fn;x;a] =
  [atom[fn] --> [eq[fn;CAR] --> cadar[x];                                *
                 eq[fn;CDR] --> caddar[x];                               *
                 eq[fn;CONS] --> cons[PAIR;                              *
                                      cons[car[x];cons[cadr[x];nil]]];  *
                 eq[fn;ATOM] --> atom[cadar[x]];                         *
                 eq[fn;EQ] --> eq[cadar[x];cadadr[x]];                   *
                 T --> apply[eval[fn;a];x;a]];
   eq[car[fn];LAMBDA] --> eval[caddr[fn];pairlis[cadr[fn];x;a]];
   eq[car[fn];LABEL] --> apply[caddr[fn];x;
                               cons[cons[cadr[fn];caddr[fn]];a]]

eval[e;a] = [atom[e] --> cdr[assoc[e;a]];
      atom[car[e]] -->
               [eq[car[e];SYMBOL] --> e;                                 *
                eq[car[e];NIL] --> e;                                    *
                eq[car[e];PAIR] --> e;                                   *
                eq[car[e];COND] --> evcon[cdr[e];a];
                T --> apply[car[e];evlis[cdr[e];a];a]];
      T --> apply[car[e];evlis[cdr[e];a];a]]
```

Figure 5: The modified universal functions. The auxiliary functions such as *evcon* must also be modified to untag arguments appropriately.

$$\begin{tikzcd}
\texttt{(QUOTE (LAMBDA (X) X))} \arrow[r, "Mention"'] & \texttt{(LAMBDA (X) X)} \arrow[r, "Mention"'] & \lambda[[x];x] \\
Use \arrow[u] & Use \arrow[u] &
\end{tikzcd}$$

Figure 6: Higher-Order Functions in S-expression LISP.

evaluating

$$\mathcal{R}(mapcar[\mathcal{R}(\lambda[[x];x]);lst]) \equiv \texttt{(MAPCAR (QUOTE (LAMBDA (X) X)) LST)}$$

(See Figure 6.) The modified evaluator halts in an error state given the corresponding inductive representation as the reader may verify.

In a similar way one can show the disabling effect of the inductive coding on the *eval* operator and on macros. For example, under the new coding (3) becomes

```
(cons (symbol car)
      (pair (pair (symbol quote)
                  (pair (pair (symbol a) (symbol b)) (nil))) (nil)))
```

Since the arguments are constants its value is simply

```
(pair (symbol car)
      (pair (pair (symbol quote)
                  (pair (pair (symbol a) (symbol b)) (nil))) (nil)))
```

and an explicit application of *eval* has no effect.

It is also worth noting the connection between S-expression LISP's dynamic representation decoding and the previously cited "well-known" problems arising from the LISP 1.5 evaluator's iterative evaluation of function variables and dynamic scoping: neither phenomena would have existed had the representation map $\mathcal{R}$ been a well-defined induction.

The metalinguistic power arising from (12) is achieved at the cost of pervasive confusion in LISP between program and data.[4] Nevertheless, the trade-off seems to have been essential in S-expression LISP since it is unlikely that the language would have attracted much interest without facilities for defining higher-order functions such as *mapcar*.

But what is the effect of the inductive coding on the S-expression dialect Scheme? Its abstract syntax, Figure 7, can be obtained by simplifying that of the original representation language S-expression LISP depicted in Figure 4. As in S-expression LISP the inductive version of Scheme has a simple and intuitively reasonable λ-calculus-like operational semantics. On the face of it this seems quite reasonable since Scheme correctly avoids the use of quotation in expressing higher-order functions. But higher-order functions in S-expression LISP are only one of a number of desirable programming language features arising from

[4]Wadler [Wad87], for example, notes that when students were asked to determine the value of the expression `(car (quote (a . b)))` some responded that the value was `quote`, some that the value was the symbol `a`, and still others that it was the value of the variable named by the symbol `a` whatever that might be.

$$\begin{array}{lll}e & ::= & (\mathrm{quote}\ s)\ \mid\ x\ \mid\ (e\ ...\ e)\ \mid\ (\mathrm{lambda}\ (x\ ...\ x)\ e)\ \mid\ (\mathrm{if}\ e\ e\ e) & \text{— Forms}\\ s & ::= & x\ \mid\ ()\ \mid\ (s\ .\ s) & \text{— S-expressions}\end{array}$$

Figure 7: The Abstract Syntax of Scheme.

(12) and we are left with the same puzzling question posed in the Introduction: How is a structured operational semantics for Scheme to be reconciled with its powerful *expression macros* or *eval* operator[5]?

Having identified the common source of these features, we propose a systematic treatment. First, the practice of programming in a representation language inevitably leads to confusion between notation, representation and object. It ought to be avoided. Function values are best expressed explicitly in the language syntax and macros can be expressed without any reference to representation by adhering to a pattern matching discipline as suggested in [KW87]. This leaves only *eval* and reification. As we show in Section 4, these operators can be reintroduced in an otherwise representation-independent dialect. Moreover, since these are the only operators which require dynamic conversion between programs and their representations, we can encapsulate the conversions within these operators thereby insulating the vast majority of non-metalinguistic applications from the conversion's widely noted ill-effects (cf., [Smi84, FW84, Wad87, Ada]).

3 M-LISP

Motivated by the concerns described in the preceding section we are led to propose the following hybrid of the abstract syntaxes of M-expression LISP and Scheme:

$$M ::= X\ \mid\ []\ \mid\ [M\ .\ M]\ \mid\ x\ \mid\ (\lambda x.M)\ \mid\ (M\ M)\ \mid\ (\texttt{IF}\ M\ M\ M)$$

We have arrived at the syntax of the familiar untyped λ-calculus extended with constants[6], a constructor for pairs and a conditional. This is reassuring since it conforms not only with our intuitive understanding of Pure LISP but also with its denotational semantics which abstracts away from the operational issues in question here.

The abstract syntax is not idealized as is often the case for studies of LISP. M-LISP is fully curried. Functions of several arguments, like proper lists, are expressed in concrete syntax:

$$\begin{array}{rcl}[M_1\ M_2\ \ldots\ M_n] & \equiv & [M_1\ .\ [M_2\ .\ [\ \ldots\ [M_n\ .\ []]\ \ldots\ ]]]\\ (\lambda x_1 x_2\ \ldots\ x_n.M) & \equiv & (\lambda x_1.(\lambda x_2.(\ \ldots\ (\lambda x_n.M)\ \ldots\ )))\\ (M_1\ M_2\ M_3\ \ldots\ ) & \equiv & ((M_1\ M_2)\ M_3\ \ldots\ ).\end{array}$$

In the remainder of the paper we will omit the conditional but all the results go through when it is included.

[5]Although it is not specified in the standard, all Scheme implementations known to the author include an *eval* operator.

[6]For the purposes of this paper we follow McCarthy's original convention of distinguishing constants from variables by case. In practice, we only require that the constants be distinguished from variables in some manner (e.g., by different fonts.)

We will now develop M-LISP's operational semantics. This will be based on a reduction relation which relates programs to their values. We will then derive the calculus for reasoning about operational equivalence between terms.

3.1 Preliminaries

We begin by defining some notation. We generally adopt the terminology and notation of [Bar84]. We use the symbol Λ_π to denote the set of terms, the metavariables M, N, ... to range over terms, the metavariables x, y, ... to range over variables and the metavariable X to range over symbolic constants. This set includes the boolean constants `TRUE` and `FALSE`. For pure LISP the terms *constant*, *atom* and *symbol* are interchangeable. The set of *free variables* of a term $FV(M)$ is defined by induction on their structure in the usual way. We use the notation A^B to denote the set of terms A with free variables drawn from B. $\Lambda_\pi^\emptyset$ denotes the set of closed terms. We use the symbol $\equiv$ to denote syntactic equality. Since we identify terms that differ only in the names of their bound variables we may write $(\lambda x.x) \equiv (\lambda y.y)$.

A *value* is any term of the form X, x, $[]$, $(\lambda x.M)$ or $[M_1 \;.\; M_2]$ whenever M_1 and M_2 are values. We use the symbol V for the subset of Λ_π of values. The symbol $V^\emptyset$ denotes the set of closed values. A *program* is a closed Λ_π-term.

The substitution of M for free occurrences of x in N, denoted $N[x := M]$, is given by:

$$\begin{array}{rcll}
X[x := M] & \equiv & X & \\
[][x := M] & \equiv & [] & \\
[N \;.\; L][x := M] & \equiv & [N[x := M] \;.\; L[x := M]] & \\
x[x := M] & \equiv & M & \\
y[x := M] & \equiv & y & y \not\equiv x \\
(N\ L)[x := M] & \equiv & (N[x := M]\ L[x := M]) & \\
(\lambda x.N)[x := M] & \equiv & (\lambda x.N) & \\
(\lambda y.N)[x := M] & \equiv & (\lambda z.N[y := z][x := M]) & y \not\equiv x,\ z \text{ unique.}
\end{array}$$

The *program contexts* $C\{\}$ are given by

$$C\{\} ::= X \mid x \mid [] \mid \{\} \mid [C\{\} \;.\; C\{\}] \mid (\lambda x.C\{\}) \mid (C\{\}\ C\{\}). \qquad (13)$$

3.2 Operational Semantics

In Figure 8 we define the *leftmost* one-step relation $\xrightarrow[v]{}$. This defines a partial function which gives the operational semantics of M-LISP. As usual, $\xrightarrow[v]{}\!\!\!\!\rightarrow$ denotes the reflexive and transitive closure of $\xrightarrow[v]{}$. The evaluator is then given by

Definition 1 (Eval) *Eval*$(M) = N$ iff $M \xrightarrow[v]{}\!\!\!\!\rightarrow N$ for program M and value N.

The evaluator naturally gives rise to a notion of equivalence between terms. Two terms are operationally equivalent if one can be replaced by the other without changing the outcome of the computation on observable values. Here symbols are the observable values.

Axioms:

$$\begin{array}{rcll}
((\lambda x.M)\ N) & \xrightarrow[v]{} & M[x := N] & \text{— } N \in V \\
(\texttt{CAR}\ [M_1\ .\ M_2]) & \xrightarrow[v]{} & M_1 & \text{— } M_1, M_2 \in V \\
(\texttt{CDR}\ [M_1\ .\ M_2]) & \xrightarrow[v]{} & M_2 & \text{— } M_1, M_2 \in V \\
(\texttt{EQ?}\ M_1\ M_2) & \xrightarrow[v]{} & \texttt{TRUE} & \text{— } M, N \text{ symbols}, M \equiv N \\
(\texttt{EQ?}\ M_1\ M_2) & \xrightarrow[v]{} & \texttt{FALSE} & \text{— } M, N \text{ symbols}, M \not\equiv N \\
(\texttt{ATOM?}\ M) & \xrightarrow[v]{} & \texttt{TRUE} & \text{— } M \in V, \text{ a symbol} \\
(\texttt{ATOM?}\ M) & \xrightarrow[v]{} & \texttt{FALSE} & \text{— } M \in V, \text{ not a symbol}
\end{array}$$

Inference rules:

$$\frac{M \xrightarrow[v]{} M'}{(M\ N) \xrightarrow[v]{} (M'\ N)}$$

$$\frac{N \xrightarrow[v]{} N', M \in V}{(M\ N) \xrightarrow[v]{} (M\ N')} \tag{14}$$

$$\frac{M \xrightarrow[v]{} M'}{[M\ .\ N] \xrightarrow[v]{} [M'\ .\ N]},\ \frac{N \xrightarrow[v]{} N', M \in V}{[M\ .\ N] \xrightarrow[v]{} [M\ .\ N']}$$

Figure 8: The Operational Semantics of Pure M-LISP.

Definition 2 (Observational Congruence) Two terms $M, N \in \Lambda_\pi$ are *observationally congruent*, $M \simeq N$, if for all contexts $C\{\}$ such that $C\{M\}$ and $C\{N\}$ are closed, either both Eval($C\{M\}$) and Eval($C\{N\}$) are undefined or they are both defined and one is a particular symbol if and only if the other one is.

3.3 The Calculus

We now develop the call-by-value calculus λ_{π_v}.[7] We will consider several binary relations over Λ_π. The fundamental relations are *notions of reduction* R. These will be specified with set notation:

$$R = \{(M, N) \mid \text{conditions}\}.$$

$M \in \Lambda_\pi$ is an *R-redex* if for some $N \in \Lambda_\pi, (M, N) \in R$. A term is in (call-by-value) R-normal-form if it does not contain an R-redex.

A notion of reduction R generates three relations $\longrightarrow_R$, $\twoheadrightarrow_R$ and $=_R$ as follows:

$$(M, N) \in R \Rightarrow M \longrightarrow_R N$$

$$\frac{M \longrightarrow_R M'}{(\lambda x.M) \longrightarrow_R (\lambda x.M')}$$

$$\frac{M \longrightarrow_R M'}{(M\ N) \longrightarrow_R (M'\ N)},\ \frac{N \longrightarrow_R N'}{(M\ N) \longrightarrow_R (M\ N')}$$

[7]The λ_{π_v}-calculus is unrelated to the *surjective* pairing of [KV89].

$$\frac{M \longrightarrow_R M'}{[M\ .\ N] \longrightarrow_R [M'\ .\ N]}\ ,\ \frac{N \longrightarrow_R N'}{[M\ .\ N] \longrightarrow_R [M\ .\ N']}$$

The symbol $\twoheadrightarrow_R$ denotes the *transitive* and *reflexive* closure of $\longrightarrow_R$. If $M \twoheadrightarrow_R N$ is provable we write $R \vdash M \twoheadrightarrow N$. The symbol $=_R$ denotes the *symmetric and transitive closure* of $\twoheadrightarrow_R$ and defines the congruence relation of R. If $M =_R N$ is provable then we write $R \vdash M = N$.

We will require the following definitions. definition A notion of reduction R on Λ_π is *substitutional* if for all M, N, L and x, $(M, N) \in R \Rightarrow (M[x := L], N[x := L]) \in R$.

definition A notion of reduction R is *Church-Rosser* (*CR* or *confluent*) if $\twoheadrightarrow_R$ has the diamond property. That is,

$$\forall M, M_1, M_2 [M \twoheadrightarrow_R M_1, M \twoheadrightarrow_R M_2] \Rightarrow \exists M_3 [M_1 \twoheadrightarrow_R M_3, M_2 \twoheadrightarrow_R M_3].$$

Several of the proofs of confluence in the paper employ the method of commutative diagrams. Let $\longmapsto$ denote an (arbitrary) binary relation on a set S. We use the symbol $\overset{=}{\longmapsto}$ for the *reflexive closure* of $\longmapsto$ and $\overset{=}{\longmapsto}^*$ for the *reflexive* and *transitive* closure of $\longmapsto$. definition Let $\longmapsto_1$ and $\longmapsto_2$ be two binary relations on a set S. Then $\longmapsto_1$ *commutes with* $\longmapsto_2$ if $\forall M$, M_1, $M_2 \in S$, such that $M \longmapsto_1 M_1$ and $M \longmapsto_2 M_2 \Rightarrow \exists M_3 \in S$ such that $M_1 \longmapsto_2 M_3$ and $M_2 \longmapsto_1 M_3$.

The following two general lemmas [Ros73] will be used extensively throughout the paper.

Lemma 1 (Rosen) *Let $\longmapsto_1$ and $\longmapsto_2$ be binary relations on a set S. Suppose that $\forall M$, M_1, $M_2 \in S$, $M \longmapsto_1 M_1$, $M \longmapsto_2 M_2 \Rightarrow \exists M_3 \in S$, such that $M_1 \overset{=}{\longmapsto}_1 M_3$ and $M_2 \overset{=}{\longmapsto}_1^* M_3$, then $\overset{=}{\longmapsto}_1^*$ commutes with $\overset{=}{\longmapsto}_2^*$.*

Lemma 2 (Hindley-Rosen) *Let R_1 and R_2 be CR notions of reduction on a set S such that $\twoheadrightarrow_{R_1}$ commutes with $\twoheadrightarrow_{R_2}$. Then $R_1 \cup R_2$ is CR.*

definition

1. Let R and R' be notions of reduction on sets A and B respectively. R' is an *extension* of R if

   (a) $A \subseteq B$.

   (b) $\forall a$, $a' \in A$, $a \longrightarrow_R a' \Leftrightarrow a \longrightarrow_{R'} a'$.

   (c) $\forall a \in A$, $a \longrightarrow_{R'} a' \Rightarrow a' \in A$.

2. Let R, R' be as in 1. R' is a *conservative extension* of R if $\forall a$, $a' \in A$, $a =_R a' \Leftrightarrow a =_{R'} a'$.

Proposition 1 *Any confluent extension is conservative.*

$$\begin{array}{rcl} car & = & \{((\texttt{CAR}\ [M\ .\ N]), M) \mid M, N \in V\} \\ cdr & = & \{((\texttt{CDR}\ [M\ .\ N]), N) \mid M, N \in V\} \end{array}$$

$$eq? = \{((\texttt{EQ?}\ M\ N), \begin{cases} \texttt{TRUE} & \mid M, N \text{ symbols}, M \equiv N \\ \texttt{FALSE} & \mid M, N \text{ symbols}, M \not\equiv N \end{cases})\}$$

$$atom? = \{((\texttt{ATOM?}\ M), \begin{cases} \texttt{TRUE} & \mid M \in V,\ \text{an atom} \\ \texttt{FALSE} & \mid M \in V,\ \text{not an atom} \end{cases})\}$$

Figure 9: The Primitive Notions of Reduction.

The λ_{π_v}-calculus is induced by the call-by-value notion of reduction β_{π_v} over Λ_π.

$$\beta_{\pi_v} = \{((\lambda x.M)N, M[x := N]) \mid M \in \Lambda_\pi, N \in V\}.$$

The consistency of the equational theory λ_{π_v} follows in the usual way from the following lemma.

Lemma 3 *The notion of reduction β_{π_v} is CR.*

Proof: The proof employs a parallel reduction relation following Tait/Martin-Löf.
Remark: It is unsurprising that the notion of reduction $\beta_\pi = \{((\lambda x.M)N, M[x := N]) \mid M, N \in \Lambda_\pi\}$ is also *CR*.

The reduction relations over Λ_π which define the remaining McCarthy primitives {*car*, *cdr*, *eq?*, *atom?*} are given in Figure 9. We use *prim* as a metavariable over this set. When no confusion arises we write the name of the notion of reduction in place of the constant in the relation.

Lemma 4 *Let prims = car $\cup$ cdr $\cup$ eq? $\cup$ atom?. The notion of reduction prims is CR.*

Proof: (Sketch) First we show that each of the notions of reduction *prim* is *CR*. This follows by the fact that each of the $\overset{=}{\longrightarrow}_{prim}$ relations has the diamond property. Then the proof bootstraps as follows. By lemma 2 the union of commutative *CR* reduction relations is *CR*. It is shown by cases of *car* and *cdr*-redexes that $\twoheadrightarrow_{car}$ and $\twoheadrightarrow_{cdr}$ commute. Then it is shown that $\twoheadrightarrow_{car \cup cdr}$ and $\twoheadrightarrow_{eq?}$ commute. Finally, we show that $\twoheadrightarrow_{car \cup cdr \cup eq?}$ and $\twoheadrightarrow_{atom?}$ commute.
Remark:

1. M-LISP does not have a primitive *cons* operator since construction is syntactic.
2. The constants *eq?* and *atom?* can also be given by Plotkin's *Constapply*:

$$Constapply : Constants \times Constants \to Closed\ Values$$

   This is not the case for projections *car* and *cdr* since their domains include arbitrary values.

1. Any variable x, constant X or $[]$ is an s.r. sequence.
2. If $N_2, \ldots, N_k$ is an s.r. sequence and $N_1 \xrightarrow[v]{} N_2$ then $N_1, N_2, \ldots, N_k$ is an s.r. sequence.
3. If $N_1, \ldots, N_k$ is an s.r. sequence then $(\lambda x_1.N_1), \ldots, (\lambda x_k.N_k)$ is an s.r. sequence.
4. If $M_1, \ldots, M_j$ and $N_1, \ldots, N_k$ are s.r. sequences then so are:

$$(M_1\ N_1), \ldots, (M_j\ N_1), \ldots, (M_j\ N_k) \text{ and } [M_1\ .\ N_1], \ldots, [M_j\ .\ N_1], \ldots, [M_j\ .\ N_k].$$

Figure 10: Standard Reduction Sequences.

Theorem 1 (CR for M-LISP) *Let mlisp* $= \beta_{\pi_v} \cup prims$. *The notion of reduction mlisp is CR.*

Proof: (Sketch) The proof is by commutative diagrams. By lemma 1 it suffices to show the following

$$\forall M, M_1, M_2[M \longrightarrow_{\beta_{\pi_v}} M_1, M \longrightarrow_{prims} M_2] \Rightarrow \exists M_3[M_2 \overset{\equiv}{\longrightarrow}_{\beta_{\pi_v}} M_3, M_1 \twoheadrightarrow_{prims} M_3].$$

This is shown by cases and by induction on the structure of terms.

Corollary 1 *The equational logic of pure M-LISP is consistent.*

Having established the consistency of the equational logic we now define a standard reduction sequence which gives a (call-by-value) normal order reduction sequence for M-LISP and provides the link between the logic and the operational semantics.

Definition 3 (Standard Reduction Sequence) A *standard reduction sequence* (s.r. sequence) is defined in Figure 10.

The connection between the operational semantics and the calculus can now be given in terms of a standardization theorem and a corollary. The proofs of these and the Correspondence theorem are adaptations of Plotkin's proofs in [Plo75].

Theorem 2 (Standardization)
mlisp $\vdash M \twoheadrightarrow N$ *iff there exists an s.r. sequence* $M', \ldots, N'$, *such that* $M \equiv M'$, *and* $N \equiv N'$.

Corollary 2 *mlisp* $\vdash M \twoheadrightarrow N$ *for some value* N *iff* $M \underset{v}{\twoheadrightarrow} N'$ *for some value* N'.

Theorem 3 (Correspondence) *mlisp* $\vdash M = N \Rightarrow M \simeq N$.

This concludes our development of the operational semantics and equational logic of pure M-LISP.

Figure 11: Representations of Terms.

4 Extended M-LISP

We now extend M-LISP with the metalinguistic *eval* and *reify* operators. These are segregated from pure M-LISP for several reasons. First, they compromise M-LISP's representation independence. In the extended language a canonical representation of terms will be required which serves as an interface between the application and the metalanguage. Second, the features cannot be implemented efficiently. Like its S-expression LISP counterpart, M-LISP's *eval* operator is not compositional [MP80]. *Reification* is also inherently interpretive. So it will not be possible to develop an efficient compiler for the extended language.

4.1 Representation of Terms

Consider the LISP expression `(eval (quote (car (quote (a . b)))))`. The argument evaluates to the list `(car (quote (a . b)))` and the second round of evaluation induced by the explicit application of *eval* results in the constant `a`. In our reduction setting, on the other hand, if $N \in V$ then for no N' is it the case that $N \xrightarrow[v]{} N'$. So it is not clear how a second round of evaluation is to have any effect.

The key aspect of the evaluation in S-expression LISP is quote's representation decoding which shifts between the mention and the use of

$$\texttt{(car (quote (a . b)))} \equiv \mathcal{R}(car[\texttt{(A . B)}])$$

That this shift takes place across-the-board *before* the application of *eval*, however, is of no consequence. The same effect can be achieved if the shift occurs after its application. Analogous remarks may be made about the reification operator which provides explicit access to the representation of a term.

Accordingly, M-LISP's interfaces between the language and metalanguage — *eval* and *reify* — will localize the shifting between levels of representation which are performed globally in S-expression LISP by the *quote* form. Specifically, the *eval* operator will convert representations of terms to terms. This shifts from mention to use. *Reify* will convert terms to their representations. This shifts from use to mention.

We will use the symbol $\bar{\Lambda}_\pi$ to denote the subset of closed call-by-value normal-forms which serve as representations of terms. It is natural to define $\bar{\Lambda}_\pi$ by induction on the structure of terms:

$$\bar{\Lambda}_\pi = \{\mathcal{R}(M) \mid M \in \Lambda_\pi\} \subset V^\emptyset \subset \Lambda_\pi,$$

where $\mathcal{R}$ is any (total) bijective function from $\Lambda_\pi \to \bar{\Lambda}_\pi$. So there exists a total $\mathcal{R}^{-1} : \bar{\Lambda}_\pi \to \Lambda_\pi$ such that $\forall M \in \Lambda_\pi, M \equiv \mathcal{R}^{-1}(\mathcal{R}(M))$ and $\forall \bar{M} \in \bar{\Lambda}_\pi, \bar{M} \equiv \mathcal{R}(\mathcal{R}^{-1}(\bar{M}))$. This gives a natural isomorphism between terms and their representations.
Remark:

$$\begin{array}{rcl}
\mathcal{R}(X) & \equiv & \texttt{[SYMBOL } X\texttt{]} \\
\mathcal{R}(\texttt{[]}) & \equiv & \texttt{[NIL]} \\
\mathcal{R}(x) & \equiv & \texttt{[IDENT } X\texttt{]} \\
\mathcal{R}(\texttt{[}M_1 \texttt{ . } M_2\texttt{]}) & \equiv & \texttt{[PAIR } \mathcal{R}(M_1)\ \mathcal{R}(M_2)\texttt{]} \\
\mathcal{R}((M_1\ M_2)) & \equiv & \texttt{[APP } \mathcal{R}(M_1)\ \mathcal{R}(M_2)\texttt{]} \\
\mathcal{R}((\lambda x.M)) & \equiv & \texttt{[ABS } \mathcal{R}(x)\ \mathcal{R}(M)\texttt{]}
\end{array}$$

Figure 12: The Standard Representation of Terms.

1. The invertibility of $\mathcal{R}$ ensures that reification will be complimentary to evaluation (See proposition 2) and the requirement that $\mathcal{R}(M)$ is closed insures that dynamic introductions of higher-order representations will not be subject to captures of free variables.

2. The bijectivity condition on $\mathcal{R}$ is not met by the representation defined in Figure 3. For example, the list `(CAR (QUOTE (A . B)))` has two representations,

   `(QUOTE (CAR (QUOTE (A . B))))` and `(CAR (QUOTE (A . B)))`

   and the latter is also the representation of the application *car*[`(A . B)`].

In Figure 12 we define a *standard representation* by induction on the structure of terms. The intention is that the representation satisfy the usual properties of an abstract syntax [McC63]. Like LISP's original definition it maps terms to first-order symbolic representations. It is easy to see that the standard representation satisfies the conditions above.

4.2 The Eval Operator

4.2.1 Operational Semantics

The operational semantics of the *eval* operator can now be defined by extending the leftmost relation $\underset{v}{\rightarrow\!\!\!\!\rightarrow}$ of Figure 8 to include a new axiom:

$$(\texttt{EVAL } \mathcal{R}(M)) \underset{v}{\rightarrow\!\!\!\!\rightarrow} M \qquad\qquad — M \in \Lambda_\pi^\emptyset.$$

This provides for the dynamic construction and subsequent execution of programs. For example, using the standard representation defined in Figure 12, we have

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{eval}\ ((\lambda x.\texttt{[ABS [IDENT Y] [}x\texttt{ Y]]})\ \texttt{IDENT})\ \texttt{A}} \\
\underset{v}{\rightarrow\!\!\!\!\rightarrow} & \mathit{eval}\ \texttt{[ABS [IDENT Y] [IDENT Y]] A} \\
\underset{v}{\rightarrow\!\!\!\!\rightarrow} & (\lambda y.y)\ \texttt{A} \\
\underset{v}{\rightarrow\!\!\!\!\rightarrow} & \texttt{A}.
\end{array}$$

Remark:

1. Unfortunately, there is a mismatch between the above defined operational semantics and that of S-expression LISP's *eval* operator — the latter can produce open terms while the former cannot. The reason for the restriction is explained shortly.

2. Readers familiar with S-expression LISP's *backquote* form will recognize the similarity between backquoted expressions and M-LISP pairs whose components contain redexes or variables.

3. The operational semantics introduced here is idealized. In practice calls to *eval* usually contain representations of expressions which include *macro calls* such as the `LET` form. These forms must be syntactically transcribed to a core form before the form is reduced.

4.2.2 Equational Logic

As a notion of reduction it is natural to define *eval* as follows:

$$eval = \{((\texttt{EVAL}\ \mathcal{R}(M)), M) \mid M \in \Lambda_\pi^{\emptyset}.\} \tag{15}$$

For simplicity we shall restrict our attention to the interaction of *eval* with the notion of reduction β_{π_v} (i.e., without the McCarthy constants).

Lemma 5 *The notion of reduction* eval *is CR.*

Proof: A term $(\texttt{EVAL}\ M) \in \Lambda_\pi$ is an *eval*-redex only when $M \in \bar{\Lambda}_\pi$. By definition, the elements of $\bar{\Lambda}_\pi$ are closed call-by-value normal-forms. Thus, *eval*-redexes are never nested.

By the closure conditions in $\mathcal{R}$ and in (15) we have

Lemma 6 *The notion of reduction* eval *is substitutional.*

Theorem 4 *The notion of reduction* $\text{eval} \cup \beta_{\pi_v}$ *is CR.*

Proof: By lemmas 1, 2, 3, and 5, it suffices to show the following.

$$\begin{tikzcd} & M \arrow[dl, "\beta_{\pi_v}"'] \arrow[dr, "eval"] & \\ M_1 \arrow[dr, "eval"'] & & M_2 \arrow[dl, "\xrightarrow{=}_{\beta_{\pi_v}}"] \\ & M_3 & \end{tikzcd}$$

This is shown by induction on the structure of M using lemma 6.
Remark:

1. The closure condition on M is essential. The notion of reduction corresponding to S-expression LISP's *eval* operator:

$$\{((\texttt{EVAL}\ \mathcal{R}(M)), M) \mid M \in \Lambda_\pi\} \cup \beta_{\pi_v}$$

is neither substitutional nor CR. A counter-example is the term:

$$\begin{tikzcd}[column sep=small] & (\lambda x.eval\ \texttt{[IDENT X]})\ y \arrow[dl] \arrow[dr] & \\ x & & y \end{tikzcd}$$

It is then straightforward to show that equational system induced by the extended calculus is inconsistent.

2. If $\mathcal{R}(M)$ were a closed value, say, rather than a closed call-by-value normal-form, then either *eval* would not be CR or $\mathcal{R}$ would not be bijective. For example,

$$\begin{tikzcd}[column sep=small] & eval\,\mathcal{R}(M) \arrow[dl] \arrow[dr] & \\ M & & eval\,\mathcal{R}(M)' \end{tikzcd}$$

3. Theorem 4 also follows from the following classical result:

Theorem 5 (Kleene, 1936) *Let $\bar{M} \in \Lambda$ denote the Church-numeral of the Gödel number of M. Then $\exists E \in \Lambda^{\emptyset}, \forall M \in \Lambda^{\emptyset} : E\ \bar{M} =_{\beta} M$.*

But the axiom captures the crucial property of the stored-program model: that program M and data $\mathcal{R}(M)$ are represented by the very same bits in the store.

Corollary 3 *The equational logic of* eval $\cup\ \beta_{\pi_v}$ *is consistent.*

Corollary 4 Mlisp $\cup$ eval *is a conservative extension of* mlisp.

Moreover, the standardization and correspondence theorems go through as expected. Thus, the extension of M-LISP with an explicit *eval* operator encounters no insurmountable difficulties on the logical side.

4.3 The Reify Operator

4.3.1 Operational Semantics

We now consider reification. The operational semantics of reifiers is obtained by extending the leftmost relation $\xrightarrow[v]{}$ of Figure 8 with the axiom:

$$(\texttt{REIFY}\ M) \xrightarrow[v]{} \mathcal{R}(M) \qquad\qquad — M \in \Lambda_{\pi}^{\emptyset}.$$

The intended behavior of the operator is to yield the standard representation of its *un-evaluated* argument. Since LISP uses a call-by-value reduction strategy, rule (14) must be changed so that the argument is not evaluated. We replace (14) with the following rule,

$$\frac{N \xrightarrow[v]{} N', M \in V \not\equiv \texttt{REIFY}}{(M\ N) \xrightarrow[v]{} (M\ N')} \tag{16}$$

Thus, the argument is not reducible until the function position has reduced to a value other than `REIFY`. For example, we have

$$(\textit{reify}\ (\textit{car}\ \texttt{[A . B]}))\xrightarrow[v]{}\ \texttt{[APP [IDENT CAR] [PAIR [SYMBOL A] [SYMBOL B]]]}$$

Remark:

1. Unfortunately, we have again been forced to introduce a mismatch between the operational semantics and the actual behavior of reification in S-expression LISP which provides for reification of open terms. It is plain to see that the mismatch parallels that introduced by the closure condition for the *eval* operator and, as we will see, it is imposed for the same reason.

2. Reification can also be introduced in a form more familiar to those acquainted with fexprs by extending the abstract syntax of terms to include abstractions tagged in the style of InterLISP's *named lambda*:

$$M ::= \ldots \ \mid\ (\lambda x_r.M).$$

The subscript on the binding occurrence indicates that any corresponding applied occurrence will be replaced by the representation of the argument: $(\lambda x_r.M)N \rightarrow M[x := \mathcal{R}(N)]$. We have chosen the formalism above to keep separate the notions of reification and substitution. The latter approach can easily be obtained from the former by,

$$((\lambda x_r.M)\ N) \equiv ((\lambda x.M)\ (\textit{reify}\ N)).$$

Let $\xrightarrow[v]{}\!\!\!\rightarrow$ now denote the operational semantics obtained by adding the *eval* axiom to the above described reifying extension of M-LISP. The connection between the *eval* and *reify* operators is guaranteed by the conditions on $\mathcal{R}$.

Proposition 2 *For closed* $M \in \Lambda_\pi$, $(\text{eval (reify } M))\xrightarrow[v]{}(\text{eval } \mathcal{R}(M))\xrightarrow[v]{}M.$

In our example then we have:

$$\begin{array}{lll} \textit{eval}\ (\textit{reify}\ (\textit{car}\ \texttt{[A . B]})) & \xrightarrow[v]{} & \textit{eval}\ \texttt{[APP [IDENT CAR] [PAIR [SYMBOL A] [SYMBOL B]]]} \\ & \xrightarrow[v]{} & \textit{car}\ \texttt{[A . B]} \\ & \xrightarrow[v]{} & \texttt{A} \end{array}$$

as in S-expression LISP.
Remark: It should be noted that reification provides a proper extension of the expressive power of LISP. It is well-known in λ-calculus circles that there is no $G \in \Lambda$ such that $\forall M \in \Lambda$, $G\ M =_\beta \bar{M}$, where $\bar{M}$ denotes the Church-numeral for the Gödel number of M. This can be summarized by the slogan "λ-terms cannot reflect on their own structure." Corresponding facts hold for related calculii such as that developed for M-LISP in Section 3. Thus, many functions such as the *parallel or* that cannot be written in LISP *can* be written in this extension.

4.3.2 Equational Logic

We now turn to the question of the equational logic induced by the above defined semantics. Following our approach, we wish to derive a calculus for reasoning about operational equality between programs that is, to the greatest extent possible, independent of the deterministic reduction strategy of the operational semantics. As in Section 4.2.2 we will simplify the question by restricting our attention to the interaction of the extension with a notion of β-reduction.

The following example — the dual of that in remark 4.2.2.1 of Section 4.2.2 — shows the unsurprising fact that the naive non-deterministic reifying reduction relation on open terms is not confluent.

$$\begin{tikzcd}
 & (\lambda x.\mathit{reify}\ x)\ y \arrow[dl] \arrow[dr] & \\
\mathcal{R}(x) & & \mathcal{R}(y)
\end{tikzcd}$$

Furthermore, if our equational system is to be consistent then the relations induced by the definition of program contexts (i.e., $\longrightarrow_R$, $\twoheadrightarrow_R$ and $=_R$) must be modified along the lines of (16). Otherwise there are closed terms that reduce to distinct call-by-value normal-forms. For example, let $I \equiv \lambda x.x$, and $\Delta \in \Lambda_\pi$ be a closed β_{π_v}-redex such that $\Delta \longrightarrow_{\beta_{\pi_v}} \Delta'$. We then have

$$\begin{tikzcd}
 & I\ \mathit{reify}\ \Delta \arrow[dl] \arrow[dr] & \\
\mathit{reify}\ \Delta \arrow[d] & & \mathit{reify}\ \Delta' \arrow[d] \\
\mathcal{R}(\Delta) & & \mathcal{R}(\Delta')
\end{tikzcd}$$

and it is again straightforward to show that the equational logic collapses. So we will have to settle for a weaker logic of reification in which the equality of many terms that are trivially seen to be observationally congruent is no longer provable. For example, for M, $N \in \Lambda_\pi$, and Δ as above, $M\ N\ \Delta$ is clearly observationally congruent to $M\ N\ \Delta'$, however these terms are no longer provably equal. **Remark:** From the preceding example, attempting to strengthen the logic by weakening the condition in (16) to exclude only the reification operator in the function position still leads to nonconfluence.

The schema for the new calculii is:

$$(M,N) \in R \Rightarrow M \longrightarrow_R N$$

$$\frac{M \longrightarrow_R M'}{(\lambda x.M) \longrightarrow_R (\lambda x.M')}$$

$$\frac{M \longrightarrow_R M'}{(M\ N) \longrightarrow_R (M'\ N)}\ , \quad \frac{N \longrightarrow_R N',\ M \in V \not\equiv \mathtt{REIFY}}{(M\ N) \longrightarrow_R (M\ N')}$$

$$\frac{M \longrightarrow_R M'}{[M\ .\ N] \longrightarrow_R [M'\ .\ N]}\ ,\ \frac{N \longrightarrow_R N'}{[M\ .\ N] \longrightarrow_R [M\ .\ N']}$$

A notion of reduction R now gives rise to new (restricted) derived relations $\longrightarrow_R$, $\twoheadrightarrow_R$ and $=_R$. The new notion of β-reduction is:

$$\beta'_{\pi_v} = \{((\lambda x.M)N), M[x := N] \ \mid\ M \in \Lambda_\pi, N \in V\}.$$

Lemma 7 *The notion of reduction β'_{π_v} is CR.*

The notion of reduction *reify* generating the relations $\rightarrow_{reify}$, $\twoheadrightarrow_{reify}$ and $=_{reify}$ is now given by

$$reify = \{((\texttt{REIFY}\ M), \mathcal{R}(M)) \mid M \in \Lambda^e_\pi mptyset\}. \tag{17}$$

Lemma 8 *The notion of reduction* reify *is CR.*

Proof: The confluence of $\twoheadrightarrow_{reify}$ follows from the fact that $\overset{=}{\rightarrow}_{reify}$ has the diamond property. This fact is shown by induction on M, such that $M \overset{=}{\rightarrow}_{reify} M_i$, $(i = 1, 2)$, and the well-definedness of $\mathcal{R}$. A representative case is $M \equiv PQ$. Let $\rightarrow$ abbreviate $\overset{=}{\rightarrow}_{reify}$.

Subcase 1. $P \equiv \texttt{REIFY}$ and $M_1 \equiv \mathcal{R}(Q)$. Then $Q \rightarrow Q' \Rightarrow Q' \equiv Q$. Take $M_3 \equiv M_2 \equiv \mathcal{R}(Q)$.

Subcase 2. $P \rightarrow P'$, $Q \rightarrow Q'$, $M_1 \equiv P'Q$ and $M_2 \equiv PQ'$. Take $M_3 \equiv P'Q'$.

Subcase 3. $P \rightarrow P'$, $P \rightarrow P''$, $M_1 \equiv P'Q$ and $M_2 \equiv P''Q$. Then by the induction hypothesis there exists a P''' such that $P' \rightarrow P'''$ and $P'' \rightarrow P'''$, take $M_3 \equiv P'''Q$.

Subcase 4. $Q \rightarrow Q'$, $Q \rightarrow Q''$, is the same as subcase 3.

Remark: Intuitively, the confluence of *reify* is insured by the fact that reification redexes are never nested.

Lemma 9 *The notion of reduction* reify *is substitutional.*

Proof: By the closure conditions on $\mathcal{R}$ and in (17).

Theorem 6 *The notion of reduction* reify $\cup\ \beta'_{\pi_v}$ *is CR.*

Proof: Similar to that of theorem 4, the proof proceeds by commutative diagrams using lemmas 1, 2, 7, 8 and 9.

Corollary 5 *The equational logic of* reify $\cup\ \beta'_{\pi_v}$ *is consistent.*

While we can establish the usual correspondence between the operational semantics and *reify* $\cup\ \beta'_{\pi_v}$, it is obviously not the case that *reify* $\cup\ \beta'_{\pi_v}$ is an extension (conservative or otherwise) of β_{π_v}.

5 Related Work

While there are many studies of LISP with which we share either our general methodology [Gor75, BM75, Car76, Mas86, MT89, FFKD86, Fel88], or our general subject matter [Pit80, MP80, Smi82, Smi84, RS84, FW84, FW86, DM88, Baw88], we are unfamiliar with any attempts to axiomatize LISP's metalinguistic facilities. We are also unfamiliar with any reference in the literature to the problematic nature of equation (12) in McCarthy's original representation function.

5.1 Gordon

The first structured operational semantics for LISP was defined for M-expression LISP by Gordon [Gor75]. The reduction relation induces a partial order on terms which provides a basis for an induction principle, *LISP-induction*, suitable for proving properties of non-trivial LISP programs. One application of the principle is a correctness proof for *eval*. How does this square with our claim that *eval* is essentially incorrect? The statement of the correctness condition is

Theorem 7 (Gordon) *If* $\langle e \mid a \rangle$ *is* **nice** *then* $\mathcal{E}[\![\mathrm{eval}[\mathcal{R}(e);\mathcal{R}(a)]]\!](\rho_{int}) = \mathcal{E}[\![e]\!](\mathcal{U}[\![a]\!])$.

Here e is an M-expression and a is an association list. The operational semantics reduces $\langle e \mid a \rangle$ to an S-expression value. The *nice* condition requires that the sets of function variables and value variables in e and a are disjoint. The valuation $\mathcal{E}$ defines a direct semantics. The M-expression *eval* is as defined in [McC62]. The coding $\mathcal{R}$ is McCarthy's original coding extended to code association lists. The valuation $\mathcal{U}$ produces a semantic environment ρ given an association list.

It is clear from the representations $\mathcal{R}(e)$ and $\mathcal{R}(a)$ on the left and the uncoded term e and association list a on the right, that the equation can be satisfied only if *eval* decodes. One would expect that if $\mathcal{E}[\![e]\!](\rho_{int}) = s$ then $eval[\mathcal{R}(e);\mathcal{R}(a)]$ would reduce under the operational semantics to $\mathcal{R}(s)$. But this is not the case for *eval*. Moreover, observe that the niceness condition effectively nullifies the semantic import of the representation decoding: higher-order functions.

5.2 Other Syntactic Studies of LISP

Our framework is close to that of Felleisen (et. al.,) [FFKD86, Fel88] who provide an operational characterization of Scheme's imperative control features and of assignment. However, our subject matters are different: we have not considered either the control features or assignment and they have not considered either pairs or metalinguistic features. The former but not the latter issue is explored somewhat indirectly by Revesz [Rev88] in his list oriented extension of the λ-calculus which was developed independently of our pairing calculus λ_π. Revesz's calculus is oriented more toward FP [Bac78] since its lists have an applicative property reminiscent of the construction functional form. Pfenning and Lee [PL88] have considered *eval*, but their work is carried out in a polymorphic typed setting.

5.3 2-LISP and 3-LISP

M-LISP has closer parallels to — and was originally motivated by — 2-LISP [Smi84], the precursor to the more widely noted reflective 3-LISP. In 2-LISP Smith attempts to reconcile the notions of evaluation and reduction in pure Scheme. Although it is generally agreed that this is a worthwhile goal, Smith's solution in 2-LISP was rather complicated and none of the subsequent treatments of reflection [FW84, FW86, DM88] adopted it. 2-LISP is yet another descendant of McCarthy's problematic coding algorithm but with additional structure superimposed on it.

Comparing specific aspects of M-LISP and 2-LISP, our representation function $\mathcal{R}$ corresponds to Smith's reifying *up* operator and its inverse $\mathcal{R}^{-1}$ corresponds to Smith's *down* operator. In 2-LISP down and up are user operators and down depends on the termination of its argument. Thus down is not the inverse of up. Our set $\bar{\Lambda}_\pi$ of term representations corresponds roughly to Smith's *handles*.

Extended M-LISP corresponds to a restricted form of Smith's 3-LISP and the suite of reflective languages which followed [FW84, FW86, DM88, Baw88]. The restrictions relate to the fact that our *eval* is a function of one argument: a representation of a term. 3-LISP, on the other hand, is developed in the context of an evaluator which determines the meaning of an expression relative to an environment and a continuation. 3-LISP reifiers have access not only to the representation of their argument but also to representations of the environment and continuation. In Smith's view the body of the reifier is executed "in the implementation" one level up in an "infinite tower" of evaluators. 3-LISP's *eval* operator is a function of 3-arguments: representations of an expression, an environment and a continuation and a call of *eval* spawns a new evaluation context or level of the tower below. In this paper we have made no attempt to integrate distinct contexts of evaluation into a coherent structure. There is one evaluation context and the phenomenon is developed in a purely substitutional setting.

6 Conclusions

This paper has developed two major themes:

1. We have developed a new representation-independent dialect of pure LISP based on our observation that the definition of the *quotation* form in the original definition of LISP is incorrect. In the new framework it is straightforward to define LISP's structured operational semantics and its corresponding equational logic.

2. Taking it as a given that S-expression LISP's metalinguistic *eval* and *reify* operators are desirable features, we then show how to reintroduce these constructions in the new framework. While the new forms of the operators achieve essentially the same ends as their S-expression LISP counterparts, they do so in somewhat different ways. The new operators encapsulate the representation conversions that occur globally in S-expression LISP. We then took up the question of the induced equational logics, finding that reification is introduced at the cost of equational reasoning. It would be interesting to consider a restricted form of reification which produces representations only of closed call-by-value normal-forms.

Although the simple dialect presented here is not characterized by one of LISP's "defining properties" — S-expression representation of programs — it nevertheless retains the language features usually tied to this property. This naturally raises questions about the usefulness of LISP's idiosyncratic syntactic structure. Much progress has been made along these lines in the ISWIM branch of functional languages (cf., ML, Haskell) and there is no reason that enhancements such as pattern-directed procedure invocation should not be adapted to LISP.

Finally, we hope that by shedding some light on the nature of LISP's metalinguistic power, that the present work has further reduced the noise, and thereby brought the other essential properties of LISP into sharper focus. In our view, the most important of these is LISP's dynamic typing discipline. Although Milner-style polymorphic type reconstruction has obvious advantages, it is unclear how such a strict compile-time type checking discipline is to be reconciled with very common computational situations such as file I/O that appear to be inherently dynamic.

Acknowledgements

I am happy to thank my advisor A. J. Kfoury and the other members of my committee R. S. Nikhil and Mitch Wand. I am also happy to thank Tom Cheatham, Rune Dahl, Paul Hudak, John McCarthy, Pat O'Keefe, Phil Wadler, Yuli Zhou and the anonymous referees for their many helpful suggestions. Finally, a special word of thanks to Olivier Danvy for his interest, suggestions and encouragement.

References

[Ada] D. Adams. Recognizing quote deemed harmful to eval's laziness. Private Communication.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? *Communications of The Association for Computing Machinery*, 21:613–641, 1978.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Baw88] Alan Bawden. Reification without evaluation. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 342–351, 1988.

[BM75] R. Boyer and J. Moore. Proving theorems about LISP programs. *Journal of The Association for Computing Machinery*, 22:129–144, 1975.

[Car76] R. Cartwright. User defined datatypes as an aid to verifying LISP progams. In *Automata, Languages and Programming.* Edinburgh University Press, 1976.

[DM88] Olivier Danvy and Katherine Malmkjaer. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 327–341, 1988.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 1988 ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[FFKD86] Matthias Felleisen, Dan Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proc. 1st Logic in Computer Science*, pages 131–141, July 1986.

[FW84] Dan Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 348–355, 1984.

[FW86] Dan Friedman and Mitchell Wand. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 298–307, 1986.

[Gor75] Michael Gordon. Operational reasoning and denotational semantics. Technical Report AIM-264, Stanford University, 1975.

[Kah98] G. Kahn. Natural semantics. In *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1998.

[KV89] J. W. Klop and R. De Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80, 2:97–113, 1989.

[KW87] Eugene Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 1987 ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[Mas86] Ian Mason. Equivalence of first order LISP program: Proving properties of destructive programs via transformations. In *Proc. 1st Logic in Computer Science*, pages 105–117, July 1986.

[McC61] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. In *Communications of the ACM*, pages 184–195, 1961.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[McC63] John McCarthy. Towards a mathematical science of computation. In *Proceedings of the IFIP Congress 63*, pages 21–28. North-Holland, Amsterdam, 1963.

[MP80] S. Muchnick and U. Pleban. A semantic comparision of LISP and scheme. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 56–65, 1980.

[MT89] Ian Mason and Carolyn Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Proc. 4th Logic in Computer Science*, pages 284–293, 1989.

[Mul90] Robert Muller. Syntax macros in M-LISP: A representation independent dialect of LISP with reduction semantics. Technical Report CRCT-TR-04-90, Harvard University, 1990.

[Mul91] Robert Muller. M-LISP: Its natural semantics and equational logic. In *Proceedings of the Symposium on Partial Evaluation and Semantics Based Program Manipulation, New Haven, Connecticut*, June 1991.

[Pit80] Kent Pitman. Special forms in LISP. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages 179–187, 1980.

[PL88] F. Pfenning and P. Lee. LEAP: A language with eval and polymorphism. In *Proceedings of the Conference on Theoretical Aspects of Programming and Software (TAPSOFT)*, 1988.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Rev88] G. Revesz. A list oriented extension of the church-rosser theorem. Technical Report RC 13620 (60941), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[Rey72] John Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM, 1972.

[Ros73] B. Rosen. Tree manipulation systems and church-rosser theorems. *Journal of The Association for Computing Machinery*, 20:160–187, 1973.

[RS84] J. Des Riviéres and B. Smith. The implementation of procedurally reflective languages. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 331–347, 1984.

[Smi82] Brian Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, Cambridge, MA, 1982.

[Smi84] Brian Smith. Reflection and semantics in LISP. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 23–35, 1984.

[Wad87] P. Wadler. A critique of abelson and sussman or why calculating is better than scheming. In *ACM SIGPLAN Notices*, volume 22, 3, pages 83–94, 1987.