LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Verification of Programs that Destructively Manipulate Data

Verification of Programs that Destructively Manipulate Data

by

Ian A. Mason

ECS-LFCS-87-40
(also published as CSR-248-87)

LFCS Report Series

October 1987

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1987, LFCS

Verification of Programs that Destructively Manipulate Data.

Ian A. Mason
Laboratory for Foundations of Computer Science

Edinburgh University, Edinburgh, EH9 3JZ.

Abstract

We investigate various equivalence relations between expressions in a first order functional programming language augmented with the ability to destructively alter the underlying data. To define the semantics we introduce the notion of a memory structure. A computation theory for a lexically scoped functional language is then defined over these structures. The equivalence relations are then defined within this model theoretic framework. A distinction is made between intensional relations and extensional relations. The former class turn out to have a much more manageable theory than the latter. The principal intensional relation studied is strong isomorphism, its properties allow for elegant verification proofs in a style similar to that of purely applicative languages. In particular the relation is preserved under many standard syntactic manipulations and transformations.

1. Introduction

In this paper we wish to present a framework for proving properties of programs which destructively alter their underlying data. The theory neatly separates control from data and provides a foundation for verification, derivation and transformation. In both this paper and the theory it describes we have tried to emphasize the interplay between these three areas. Indeed one of the aims of this paper is to replace the old paradigm

$$\text{Verification} = \text{Hand Simulation} + \text{Induction},$$

by one which is closer to the aims and spirit of *inferential* programming. Namely

$$\text{Verification} = \text{Transformation} + \text{Induction}.$$

To give substance to this claim we need to explain the difference between what we mean by hand simulation and transformation. At present we can only suggest the difference, after describing our framework we will be quite precise. The essence of destructive programs is that they alter the nature of existing objects, usually for the purpose of conserving time and space. Programs or expressions are then thought of as operations on the *state* or *memory* of some abstract machine. In hand simulation a program or expression is evaluated explicitly in a memory, described in a suitably abstract fashion, producing a new memory and a value therein. The task is then to show that this memory and value have certain properties, perhaps in relation to values in other memories similarly obtained. In contrast the transformation method works at the level of programs or expressions, the state or memory is never explicitly mentioned. A typical verification proof starts with one program or expression and by a sequence of simple syntactic operations, which we are calling transformations, obtains another program or expression. The resulting program or expression is thus seen, a priori, to share all the properties of the original one that are preserved by the transformations. *Inferential* programming was introduced in (Scherlis and

Scott, 1983) and emphasizes the role of derivation in programming practice. Indeed they suggest that the programs of the future will in fact be descriptions of program derivations.

We point out two other connections between verification and derivation. Firstly, on a naive level, verification of programs and derivation of programs can be viewed as *duals* of one another. In verification one proves that a given program meets or satisfies a certain specification. In derivation one does the reverse — from specifications one derives programs which accomplish the task. When such specifications are just simple-minded or well known programs that perform the required calculation or construction the connection between derivation and verification is most apparent. Program transformation is thus seen to be the vehicle of both activities. The process of derivation or verification consists in a sequence of program transformations which, presumably, preserve the appropriate extensional or intensional properties of the programs involved. This analogy, justifiably, places verification in a more favorable light than that which some have cast upon it. Its defect is that it ignores the fact that derivation, on the face of it, is a substantially more difficult task. In verification the programs are predetermined while in derivation not only are they evolving, but also the *direction* in which they are evolving changes. Also in verification one can work with an equivalence relation but in derivation the relation cannot be symmetric, rather one must take some notion of computational progress into account.

Secondly, often the verification of one program will increase the programmer's understanding to such an extent that they can write related programs that are more efficient. A good example, when the programmer is a machine, can be seen in (Goad, 1980), where the task of *specializing* an algorithm is accomplished by specializing and pruning the correctness proof of the initial algorithm. The resulting algorithm is often an order of magnitude more efficient than the one obtained by directly specializing and pruning the algorithm itself. Here the transformational approach has obvious advantages over the hand simulation approach. This is because it is at a more abstract level, and hence reveals more structure in both the algorithm and the transformations used. In the hand simulation approach it is often hard to see the forest for the trees.

One virtue of the transformation paradigm is that it emphasizes the role of transformation rather than the low-level hand simulation approach. Transformations developed and studied in the process of verification are equally applicable in the more productive process of derivation. The style is also more amenable to automation than the hand simulation variety. The dominance of the hand simulation school is largely a consequence of their preoccupation with extensional relations. To retain a transformational approach in the transition from purely applicative languages to those with side effects one must also make the transition from extensional to intensional equivalence relations.

1.1. The Underlying Data Structure

We have chosen to work in a Lisp-like world for many reasons. The most immediate is that it allows us to contrast our destructive version with the purely applicative fragment, pure Lisp, and the issues that arise in an enriched environment. More importantly, however, is that we shall, wherever possible, use pure Lisp as our specification language.

We begin by describing the underlying data structure of pure Lisp and comparing this

with that of destructive Lisp. The underlying data structure of pure Lisp, $\mathbf{S}$, is obtained from the set of atoms, $\mathbf{A}$, by closing it under a pairing operation. Thus the celebrated isomorphism: $\mathbf{S} \cong \mathbf{A} \cup (\mathbf{S} \times \mathbf{S})$. In pure Lisp the functions *car* and *cdr* are simply the first and second projection functions and are undefined on atoms. *cons* is the pairing function. *equal* determines whether two pairs are identical, i.e. whether or not their *cars* and *cdrs* are *equal*, while *atom* tells us whether the object in question is either an atom or a pair. Thus the underlying data structure of pure Lisp can be thought of as a traditional first order structure: $< \mathbf{S}, cons, car, cdr, atom, equal, \mathtt{T}, \mathtt{NIL} >$. It is thus a *static* object, in the sense that the nature of an object does not change in time. The data structure has other nice properties. For example the theory of this structure has been shown to be decidable (Tenney, 1972). Also in (Oppen, 1978) it is shown that the quantifier free part of this theory is decidable in linear time. Another important property of the data structure is that it and its important subdomains (for example lists) are built up inductively, thus allowing strong principles of induction to be used in verifying programs. One last and possibly most important principle is that Leibniz's Law[1] holds for programs; *equal* expressions can be replaced by *equal* expressions in an expression to obtain an *equal* expression:

Leibniz's Law $\quad e_0(\bar{x}) \equiv e_1(\bar{x}) \rightarrow e(\bar{x}, e_0(\bar{x})) \equiv e(\bar{x}, e_1(\bar{x}))$.

This principle has the consequence that correctness proofs in pure Lisp are very much of the *transformation* plus *induction* variety. The content of Leibniz's Law is that it lays the foundation for a calculus of program transformations. Any program that is obtained from another by replacing a portion by another Lisp equal one is guaranteed to have all the extensional properties the original one had. Thus the underlying semantics can be pushed somewhat into the background, serving merely as a *justification* for the transformations and induction principles involved.

This is not to say that pure Lisp does not have its disadvantages. The most glaring, perhaps, is the fact that it is theoretical rather than practical. It could also be argued that the simplicity of the data structure and the resulting computation theory has, perhaps, helped perpetuate the myth that there is one *single* notion of *equivalence* between programs, which is by and large an extensional notion, and as a consequence one single notion of *equivalence* preserving transformations. It also gives unjustified emphasis to *extensional* properties of programs, since the intensional relations can easily be transformed into extensional properties of related (or *derived*) programs (Talcott, 1985b). This relationship between extensional and intensional properties of programs is certainly not true in the destructive case.

In destructive Lisp we have almost exactly the opposite situation. The language is practical, efficient, but until now did not have an elegant or even nice theory surrounding it. The data domain of destructive Lisp is similar to that of pure Lisp except that it is more complex. It consists of two types of objects, atoms and cons cells. Atoms are either numbers or symbols, with two special symbols T and NIL playing the role of booleans, T for *true* and NIL for *false*. We shall usually ignore any structure $\mathbf{A}$ might have, other than containing the integers, and concentrate on the other type of object, cons cells. A cons cell is essentially an ordered pair of names or addresses of other S-expressions. These

[1] also known as referential transparency

addresses or names are usually called pointers, the first one is called the *car* pointer and the second the *cdr* pointer. This indirect reference allows for non well-founded or cyclic S-expressions, an aspect of this data structure that is becoming more and more in vogue.

The basic underlying data operations *atom*, *int*, *add1*, *sub1*, *eq*, *cons*, *car*, *cdr*, *rplaca*, *rplacd*, are easily describable with this picture in mind. *atom* is the characteristic function, using the booleans T and NIL, of the atoms. *eq* just tests whether two data objects, either cons cells or atoms, are literally identical. *cons* takes two arguments and creates a new cons cell which contains pointers to the arguments, in the order given. *car* and *cdr* just return the object pointed to by the first and second pointers in a cons cell. While the operations *rplaca* and *rplacd* destructively alter an already existing cons cell in the following fashion: given two arguments, the first of which must be a cons cell, *rplaca* will alter the contents of the first argument so that its car pointer now points to the second argument. *rplacd* similarly alters the cdr pointer. Notice that the use of *rplaca* and *rplacd* allow one to construct cons cells which point back to themselves. They also actually change the nature of existing objects. Thus these operations force us from a static model to a *dynamic* one, the S-expression memory structure. In the next section we introduce the general notion of a memory structure, the S-expression memory structure being just a particular example.

One of the problems in developing a theory for destructive Lisp is the failure of Leibniz's Law. A simple example of this is

$$cons(cons(\mathtt{T},\mathtt{T}), cons(\mathtt{T},\mathtt{T})) \equiv \mathtt{let}\{x \leftarrow cons(\mathtt{T},\mathtt{T})\}\ cons(x,x).$$

When we apply Leibniz's Law with $e(y) = rplaca(car(y), \mathtt{NIL})$ we obtain the obviously false conclusion:

$$cons(cons(\mathtt{NIL},\mathtt{T}), cons(\mathtt{T},\mathtt{T})) \equiv \mathtt{let}\{x \leftarrow cons(\mathtt{NIL},\mathtt{T})\}\ cons(x,x).$$

Thus simple syntactic manipulations, on the face of it, seem prohibited in the destructive case. This does much to explain why the vast majority of verification proofs of destructive programs in the literature are of the hand simulation variety. Thus a first step in justifying our paradigm is to recover Leibniz's Law. This is done by making the transition from extensional relations to intensional ones.

A related problem is that the operations *rplaca* and *rplacd* do not depend solely on the pointwise *isomorphism* type of their arguments. Even though x_0 and x_1 may be isomorphic objects we cannot conclude that $rplaca(x_0, y)$ and $rplaca(x_1, y)$ will return isomorphic objects. These operations not only depend on the isomorphism type of their arguments but also on how they sit together, for example if they share structure or not. Structure sharing between objects, detectable using *eq*, and cyclic or non-well-founded structures make the underlying data structure in this version of Lisp substantially more complex than in the pure case. The nature of the operations prohibit viewing the data structure, in any natural way (an example of an *unnatural* way would be to incorporate an explicit time parameter), as a traditional first order algebraic structure, rather it is a particular example of a Memory Structure, a mathematical object used to model both the *dynamic* state of various types of Random Access Memories and the operations upon them.

Another aspect where pure Lisp and destructive Lisp differ is in the richness of the control primitives. In pure Lisp the only control primitive other than function application is the branching primitive, `if`. In destructive Lisp we also require a lexical variable binding primitive, `let`, and for ease of reading a sequencing primitive `seq`. One of the main aims of the theory we are presenting here is to overcome these problems and develop a theory as mathematically elegant as the pure Lisp case.

1.2. Notation

We finish off this section by describing some of our notation. We use the usual notation for set membership and function application. Let D, D_0, D_1 be sets, $\mathsf{D}^{(n)}$ is the set of sequences of elements of D of length n. D^* is the set of finite sequences of elements of D. Some notation for sequences follows. ϵ is the empty sequence, the unique element of $\mathsf{D}^{(0)}$ for any domain D. The sequence of length n with i^{th} element d_i for $i < n$ is written $[d_0, \ldots, d_{n-1}]$ or more succinctly $\bar{d}$. $|\bar{v}|$ is the length of $\bar{v}$, while $\bar{v}\downarrow_i$ is the ith element of $\bar{v}$. $\bar{v} * \bar{u}$ is the concatenation of $\bar{v}$ and $\bar{u}$. We identify d with the singleton sequence $[d]$. Note that $(\bar{u} * \bar{v}) * \bar{w} = \bar{u} * (\bar{v} * \bar{w})$ and $[\ \ ] = \epsilon$. If $\mu \in [\mathsf{D}_0 \rightsquigarrow \mathsf{D}_1]$, the set of partial functions from D_0 to D_1, then δ_μ is the domain of μ and ρ_μ is its range. For $d_0 \in \mathsf{D}_0, d_1 \in \mathsf{D}_1$ and $\mu \in [\mathsf{D}_0 \rightsquigarrow \mathsf{D}_1]$ we let $\mu\{d_0 \leftarrow d_1\}$ be the map μ_0 such that $\delta_{\mu_0} = \delta_\mu \cup \{d_0\}$, $\mu_0(d_0) = d_1$ and $\mu_0(d) = \mu(d)$ for $d \neq d_0, d \in \delta_\mu$. Some particular sets that we shall use frequently are as follows. $\mathsf{N} = \{0, 1, 2, \ldots\}$ is the natural numbers and $n, n_0, \ldots$ range over N. We let $\mathsf{T} = \{0,1\}^*$ be the complete binary tree, i.e. the set of finite sequences of 0's and 1's. We adopt the convention that trees grow downward and $\sigma, \sigma_0, \ldots$ will range over T.

2. Memory Structures

In this section we introduce the notion of a memory structure over a set A of atoms. The purpose is to model the memory of a Random Access Machine (RAM) and to study the abstract structures typically represented in such machines. The memory of a RAM can be thought of as a collection of locations or cells (at any particular time this collection will of course be finite). The machine uses these cells or locations to store various types and quantities of objects. There are machine instructions for accessing and updating the contents of memory cells. Some objects are intended to represent abstract quantities such as numbers, boolean vectors, characters, etc., and there are machine instructions for computing functions on these abstract entities, such as arithmetic operations and boolean functions. The exact nature and number of the objects storable in each location varies from machine to machine; we shall abstract away from this machine dependent aspect of memory. Consequently we shall assume that our hypothetical machine can store a sequence of objects (the sequence being of arbitrary finite length), each object of which is either an atom from A or the address of another location in memory. An address in this sense is simply some specification of a location by which the machine can access that location (and its contents). Again the precise nature of these addresses will vary from machine to machine, and so again we shall abstract away from these implementation dependent details.

In this paper we shall be mainly concerned with S-expression memories that can only store pairs of objects in each location; however we shall treat the general case first, leaving

S-expression memory structures as a particular example. This is because the theory we develop can easily be extended to handle other data structures such as arrays, records, vectors.

Let A be some fixed set of atoms and C some countably infinite set disjoint from A. C is the set of memory cells of our hypothetical machine. The elements of the sequences that are stored in these cells are the *memory values* and we denote them by V. Thus $\mathsf{V} = \mathsf{A} \cup \mathsf{C}$. A memory μ is a function from a finite subset of C to the set of sequences of memory values, $\mathsf{V}^* = (\mathsf{A} \cup \mathsf{C})^*$. Since we wish $\mu(c)$ to represent the contents of the location c in the memory μ, we also require that those cells which occur amongst the contents of cells are also cells in our memory. Thus we define a *memory* μ to be a finite map such that

$$\mu \in [\delta_\mu \rightarrowtail (\delta_\mu \cup \mathsf{A})^*],$$

where δ_μ, the domain of μ, is a finite subset of C. The set of all memories over A and C is denoted by $\mathsf{M}_{(\mathsf{A},\mathsf{C})}$.

Now suppose that M is a set of memories. A *memory object* of M is a pair

$$[v_0, \ldots, v_{n-1}]\,;\mu$$

such that μ is a memory in M and the sequence $[v_0, \ldots, v_{n-1}]$ satisfies $v_i \in \delta_\mu \cup \mathsf{A}$ for $i < n$. Thus a memory object is a memory together with a sequence of memory values which *exist* in that memory. The reason we consider sequences of memory values and not just singletons is twofold. Firstly, we often want to apply a memory operation or defined function to several arguments all of which we assume exist in one and the same memory, and, secondly, the behavior of many of the memory operations is not determined simply by the pointwise nature of its arguments but also by how they *sit* with one another — for example, if they share structure. Hence when defining equivalence relations the pointwise approach is next to useless. A *memory structure* is defined to be a set of memories M together with a set of operations O, which are allowed to be partial, on those memory objects of M. The operations model the machine instructions for manipulating objects. We usually refer to a memory structure by its collection of memories M, taking the operations to be implicit. We also abuse notation and refer to the set of memory objects of a particular collection of memories M simply by M; context and notation should always prevent confusion. One last abuse of notation is that by $\mathsf{M}^{(n)}$ we *always* mean the collection of memory objects whose sequence of memory values is of length n. For ease of reading we let $\mu, \mu_0, \ldots$ range over memories, $v, v_0, \ldots$ range over V, $a, a_0, \ldots$ range over A and $c, c_0, \ldots$ range over C.

2.1. The S-expression Memory Structure

As a particular example of a memory structure we now present the S-expression memory structure. It should be very familiar to those readers acquainted with any Lisp-like language. We often assume that the integers Z are contained in A. A will always be assumed to contain two non-numeric atoms T and NIL, representing *true* and *false*. NIL is also used to represent the empty list. We shall also assume that there are an unlimited collection of non-numeric atoms other than the two just mentioned. We shall usually

denote them by strings of upper case letters IN THIS FONT. Thus for our purposes the following are also in $\mathbb{A}$: INFINITY, M10, THIS:ATOM, ... The set of S-expression memories, M_{sexp}, is defined by:

$$\mathsf{M}_{sexp} = \{\mu \in \mathsf{M}_{(\mathbb{A},\mathbb{C})} \mid \mu \in [\delta_\mu \rightarrow \mathsf{V}^{(2)}]\}.$$

Thus, as we mentioned earlier, the S-expression memory can only store pairs of memory values in its memory locations. It is traditional to call these *binary* cells *cons cells*. To complete our specification of the S-expression memory structure we need only describe the operations O_{sexp}.

$$\mathsf{O}_{sexp} = \{int, atom, add1, sub1, eq, cons, car, cdr, rplaca, rplacd\},$$

the definitions of which are: *int* and *atom* are characteristic functions (recognizers) of $\mathbb{Z}$ and $\mathbb{A}$, respectively. *eq* is the characteristic function of equality. $int(v\,;\mu) = \mathtt{T}\,;\mu$ if $v \in \mathbb{Z}$ and otherwise is $\mathtt{NIL}\,;\mu$. $atom(v\,;\mu) = \mathtt{T}\,;\mu$ if $v \in \mathbb{A}$ and otherwise it is $\mathtt{NIL}\,;\mu$. $eq([v_0, v_1]\,;\mu) = \mathtt{T}\,;\mu$ if $v_0 = v_1$ and otherwise it is $\mathtt{NIL}\,;\mu$. *add1* and *sub1* are the successor and predecessor functions on $\mathbb{Z}$. $add1(z\,;\mu) = z+1\,;\mu$ and $sub1(z\,;\mu) = z-1\,;\mu$. The *cons* operation is a pair constructing function and *car* and *cdr* are the corresponding projections. Note that *cons* enlarges the domain of the memory by *selecting* a new location from *free storage* and putting the arguments as its contents. The method of *selection* is of no concern to us, it is another implementation dependent feature which we abstract away. The free storage of a memory μ is just another name for $\mathbb{C} - \delta_\mu$. $cons([v_0, v_1]\,;\mu) = c\,;\mu_0$ where $c \notin \delta_\mu$ and $\mu_0 = \mu\{c \leftarrow [v_0, v_1]\}$. Assuming that $\mu(c) = [v_0, v_1]$ we have $car(c\,;\mu) = v_0\,;\mu$ and $cdr(c\,;\mu) = v_1\,;\mu$. The destructive memory operations *rplaca* and *rplacd* update the contents of a pre-existing location in memory. The domain of the resulting memory object is unchanged. By the use of these functions one can obtain memory objects that store their own locations. If $\mu(c) = [v_0, v_1]$ then $rplaca([c, v]\,;\mu) = c\,;\mu_0$ where $\mu_0 = \mu\{c \leftarrow [v, v_1]\}$, and $rplacd([c, v]\,;\mu) = c\,;\mu_0$ where $\mu_0 = \mu\{c \leftarrow [v_0, v]\}$.

2.2. Fragments of M_{sexp}

Two familiar examples of memory structures are obtained by considering the following two sets of memory operations on M_{sexp}:

$$\mathsf{O}_{pure} = \{int, atom, add1, sub1, atom{:}eq, cons, car, cdr\}$$

$$\mathsf{O}_{pure+} = \{int, atom, add1, sub1, eq, cons, car, cdr\}$$

All the operations are as defined previously except *atom:eq* which is, the characteristic function of, equality on $\mathbb{A}$: $atom{:}eq([v_0, v_1]\,;\mu) = \mathtt{T}\,;\mu$ if $v_0 = v_1$ and $v_0, v_1 \in \mathbb{A}$ otherwise it is $\mathtt{NIL}\,;\mu$ We call the memory structure with operations O_{pure} the *pure Lisp* memory structure, denoted by M_{pure}. Finally we denote the memory structure with operations O_{pure+} by M_{pure+}.

2.3. A Low Level Lisp Data Structure

Another example of a memory structure can be obtained by considering a low level implementation of Lisp, one in which cons cells have *mark* and *field bits*. Such a memory

structure is required to express and prove properties of such fundamental Lisp programs as the *mark and sweep garbage collection*, and programs it uses such as the Deutsch-Shorr-Waite marking algorithm. The mark and field bits in this version only take on two values, 0 or 1. Thus we have the following set of memories, M_{mfsexp}.

$$\mathsf{M}_{mfsexp} = \{\mu \in \mathsf{M}_{(\mathsf{A},\mathsf{C})} \mid \mu \in [\delta_\mu \rightarrow \{0,1\}^{(2)} \times \mathsf{V}^{(2)}]\}$$
$$\mathsf{O}_{mfsexp} = \mathsf{O}_{sexp} \cup \{m, setm, f, setf\}$$

Over this structure *car* and *cdr* access the third and fourth elements and m returns the value of the mark bit, while f returns the field value. *cons* returns a new cell with the mark and field bits set initially to 0. *setm* and *setf* simply update the mark and field bits, respectively. The rest of the operations are the obvious modifications.

2.4. The Path Function and *car-cdr* chains in $v\,;\mu$

There is a very simple way of regarding an element of the binary tree, T , as a *car-cdr* chain in a particular S-expression $v\,;\mu$, by adopting the convention that 0 represents the *car* direction, 1 the *cdr* direction. So for example the element $[0,1,0,1]$ of the binary tree corresponds to taking the *car* then the *cdr* then the *car* and finally the *cdr*. In most versions of Lisp this reduces to applying the single operation *cdadar*. Now given a path σ in $v\,;\mu$ it is convenient to talk about the object that lies at its end. This way of viewing paths in an S-expressions is made explicit by the following. For $v\,;\mu \in \mathsf{M}_{sexp}$ we define a partial function, called the *path* function, $\partial_{v;\mu}$ from T to V and its domain $\delta_{\partial_{v;\mu}}$ by induction on T:

$$\partial_{v;\mu}(\sigma) = \begin{cases} v & \text{if } \sigma = \epsilon, \text{ the empty word in } \mathsf{T}, \\ \mu(\partial_{v;\mu}(\sigma_0))\downarrow_i & \text{if } \sigma = \sigma_0 * i,\ i < 2 \text{ and } \partial_{v;\mu}(\sigma_0) \in \mathsf{C}. \end{cases}$$

The idea here is quite simple; a path $\sigma \in \mathsf{T}$ is in the domain of the path function iff no proper initial subpath ends in an atom. The value of the path function at a particular path, σ, in its domain is thus the element of V at its end. Thus paths that reach atoms will be terminal and those that reach cells will have exactly two immediate successor paths in the domain of the path function.

We shall refer to σ (when σ is in the domain of the path function of a memory object $v\,;\mu$) as a *car-cdr* chain in $v\,;\mu$, for the obvious reason that $\partial_{v;\mu}(\sigma)$ is the atom or cell one obtains by a suitable composition of the memory operations *car* and *cdr*. We define $\mathbf{Cells}_\mu(v)$ to be the set of cells that are reachable from $v\,;\mu$ by travelling along any *car-cdr* chain, $\{c \in \mathsf{C} \mid (\exists\sigma)(\partial_{v;\mu}(\sigma) = c)\}$. We shall often regard $\mathbf{Cells}_\mu(\bar{v})$ as a sequence rather than a set, ordered in some particular fashion. We also let $\mathbf{Cells}_\mu(\bar{v})$ be the union over $i < |\bar{v}|$ of $\mathbf{Cells}_\mu(\bar{v}\downarrow_i)$.

3. Computing over Memory Structures

Thus far we have described the underlying data structure. We now turn to the syntax of our language and its subsequent semantics. This we do in three steps. We

first define a class of expressions, E, in a lexically scoped Lisp-like language. Secondly using these expressions we specify how functions are to be defined. Finally we define a certain class of objects, called *memory object descriptions,* which are pairs consisting of an expression with no free variables, together with a memory in which every cons cell appearing in the expression exists in that memory, i.e. is in its domain. On these objects we define a *sequential reduction relation* which determines the computation that these memory object descriptions describe. The computation theory as defined in this section holds for an arbitrary memory structure; however, the reader may find it helpful to keep the S-expression memory structure in mind since it is the example that we are most interested in.

3.1. The Set of Memory Expressions

The set of expressions of our language, E, is defined as follows. Let X and F be disjoint countable sets. Elements of X are memory variable symbols and range over memory values. Elements of F are function symbols, each with an associated finite arity. We use x, x_0, ... for elements of X, f, f_0, ... for elements of F, and e, e_0, ... for memory expressions.

Definition: The set of *memory expressions* is defined inductively to be the smallest set E containing $\mathsf{V} = \mathsf{A} \cup \mathsf{C}$, X, and closed under the following formation rules: (1.) If $e_{test}, e_{then}, e_{else} \in \mathsf{E}$ then $\mathtt{if}(e_{test}, e_{then}, e_{else}) \in \mathsf{E}$. (2.) If $e_1, \ldots, e_n, e_{body} \in \mathsf{E}$ and $x_1, \ldots, x_n \in \mathsf{X}$ are distinct then $\mathtt{let}\{x_1 \leftarrow e_1, \ldots, x_m \leftarrow e_m\}e_{body} \in \mathsf{E}$. (3.) If $e_1, \ldots, e_n \in \mathsf{E}$ then $\mathtt{seq}(e_1, \ldots, e_n) \in \mathsf{E}$. (4.) If ϑ is either an n-ary memory operation or n-ary function symbol from F, and $e_1, \ldots, e_n \in \mathsf{E}$ then $\vartheta(e_1, \ldots, e_n) \in \mathsf{E}$.

Although we have allowed cells to appear in expressions, we warn the reader that this will only occur in special contexts. These contexts which will be defined shortly, are called *memory object descriptions.* The reason for these restrictions is quite simple. If we allowed arbitrary cells to appear in expressions then those expressions could only be evaluated or have meaning in a context, or more appropriately a memory, in which those cells were defined. We shall repeat this warning when we define the appropriate contexts.

The only variable binding operation is `let`. $\mathtt{let}\{y_1 \leftarrow e_1, \ldots, y_m \leftarrow e_m\}e_{body}$ binds the free occurrences of y_i to the value returned by e_i in e_{body}. The $\{y_1 \leftarrow e_1, \ldots, y_m \leftarrow e_m\}$ part of a `let` expression is called the *binding expression.* For a memory expression e the set of free variables in e, $FV(e)$, is defined in the usual manner. $e\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}$ is the result of substituting free occurrences of the y_i in e by the values v_i.

Definition: We call an expression which contains no $f \in \mathsf{F}$ or $c \in \mathsf{C}$ a *primitive term* or more often simply a *term.*

In addition to the basic constructs of our language, we also use constructs such as and, not, or and `ifn`. They are taken to be the usual Lisp abbreviations or *macros*, namely: $\mathtt{or}(e_1, e_2)$ is $\mathtt{let}\{t_1 \leftarrow e_1\}\mathtt{if}(t_1, t_1, e_2)$, $\mathtt{ifn}(e_{test}, e_{then}, e_{else})$ is $\mathtt{if}(e_{test}, e_{else}, e_{then})$. We also write $\mathtt{let}_{0 \le j \le n}\{x_j \leftarrow e_j\}e_{body}$ to denote $\mathtt{let}\{x_0 \leftarrow e_1, \ldots, x_n \leftarrow e_n\}e_{body}$ and similarly $\mathtt{seq}_{0 \le j \le n}(e_j)$ to denote $\mathtt{seq}(e_0, \ldots, e_n)$.

3.2. Function Definitions

We now make explicit what is to count as a function definition.

Definition: A system of memory function definitions D is a collection of equations of the form $\{f_0(\bar{x}_0) \leftarrow e_0, \ldots, f_n(\bar{x}_n) \leftarrow e_n\}$ that satisfies the following conditions: (1.) each $\bar{x}_i = [x_0^i, \ldots x_{m_i-1}^i]$ is a sequence, without repetitions, of variables from X of length m_i; (2.) f_i is an m_i-ary function symbol from F and $f_i \neq f_j$ when $i \neq j$; (3.) e_i must be a memory expression such that the free variables of e_i are a subset of $\bar{x}_i$; (4.) the only function symbols that occur in e_i are among $f_0, \ldots, f_n$, and (5.) no $c \in \mathsf{C}$ occurs in any of the e_i.

We are somewhat liberal in what we use as variables and function symbols, using words with suggestive names.

3.3. Memory Object Descriptions and $\gg^D$

A closed memory expression together with a *suitable* memory describes the computation of a memory object. Such pairs are called *memory object descriptions.* To make the notion of suitable precise, we fix a system of function definitions , D,

$$\{f_0(\bar{x}_0) \leftarrow e_0, \ldots, f_n(\bar{x}_n) \leftarrow e_n\}$$

Definition: A *memory object description* with respect to D is then defined to be a pair $e\,;\mu$ that satisfies the following conditions: (1.) e is a closed memory expression, i.e. e contains no free variables; (2.) any c which occurs in e is also in δ_μ; and (3.) every function symbol $f \in \mathsf{F}$ which occurs in e is defined in D.

Remark: As we have already mentioned when we defined the class of expressions, we shall make the following important convention. Only in the context of memory object descriptions will we allow expressions to contain cells; thus when we say $e(\bar{x})$ is an expression we shall implicitly be saying that no cell $c \in \mathsf{C}$ occurs in $e(\bar{x})$. It is only when we evaluate $e(\bar{x})$ at some memory object $\bar{v}\,;\mu$ that we shall allow cells to appear in e. By evaluating $e(\bar{x})$ at $\bar{v}\,;\mu$ we mean reducing the memory object description $e(\bar{v})\,;\mu$.

The basic rules for computation are given by the reduction relation $\gg^D$ on memory object descriptions. It is the least transitive relation containing the *single step* reduction relation, $\rightarrow\!>^D$, which is generated by the rules given below. The primitive cases correspond to primitive machine instructions for branching, sequencing, variable binding, execution of memory structure operations and function call, while the congruence cases are rules for reducing sub-expressions in order to reduce descriptions to primitive cases. They determine which sub-expression may be reduced and the effect of that reduction on the description containing it. We have adhered to the Lisp convention of call-by-value, and that arguments of functions are evaluated in a left to right order. The reader is reminded that $v, v_0, \ldots$ range over V, and that these are entities which cannot be further evaluated.

Definition: The single step reduction relation, $\rightarrow\!>^D$, is defined to be the smallest relation on memory object descriptions that satisfies the following two sets of rules.

- **Primitive cases:**

$$\mathtt{if}(v_0, e_{then}, e_{else})\,;\mu \rightarrow\!>^D \begin{cases} e_{then}\,;\mu & \text{if } v_0 \neq \mathrm{NIL} \\ e_{else}\,;\mu & \text{if } v_0 = \mathrm{NIL} \end{cases}$$

$$\mathtt{seq}(e)\,;\mu \rightarrow^{D} e\,;\mu$$

$$\mathtt{seq}(v_0, e_1, \ldots, e_m)\,;\mu \rightarrow^{D} \mathtt{seq}(e_1, \ldots, e_m)\,;\mu$$

$$\mathtt{let}\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}e\,;\mu \rightarrow^{D} e\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}\,;\mu$$

$$\vartheta(v_1, \ldots, v_n)\,;\mu \rightarrow^{D} v_0\,;\mu_0 \qquad \text{if } \vartheta \in \mathbf{O} \text{ and } \vartheta([v_1, \ldots, v_n]\,;\mu) = v_0\,;\mu_0$$

$$\vartheta(v_1, \ldots, v_n)\,;\mu \rightarrow^{D} e\{y_1 \leftarrow v_1, \ldots, y_n \leftarrow v_n\}\,;\mu \qquad \text{if } \vartheta(y_1, \ldots, y_n) \leftarrow e \text{ is in } D.$$

- **Congruence cases:** If $e_a\,;\mu_a \rightarrow^{D} e_b\,;\mu_b$ then

$$\mathtt{if}(e_a, e_{then}, e_{else})\,;\mu_a \rightarrow^{D} \mathtt{if}(e_b, e_{then}, e_{else})\,;\mu_b$$

$$\mathtt{seq}(e_a, \ldots)\,;\mu_a \rightarrow^{D} \mathtt{seq}(e_b, \ldots)\,;\mu_b$$

$$\mathtt{let}_{\substack{1\le i<n\\ n<j\le m}} \{y_i \leftarrow v_i, y_n \leftarrow e_a, y_j \leftarrow e_j\}e;\mu_a \rightarrow^{D} \mathtt{let}_{\substack{1\le i<n\\ n<j\le m}} \{y_i \leftarrow v_i, y_n \leftarrow e_b, y_j \leftarrow e_j\}e;\mu_b$$

$$\vartheta(v_1, \ldots, v_{j-1}, e_a, \ldots, e_m)\,;\mu_a \rightarrow^{D} \vartheta(v_1, \ldots, v_{j-1}, e_b, \ldots, e_m)\,;\mu_b$$

We say $e\,;\mu$ *evaluates* to $v_0\,;\mu_0$ if $e\,;\mu \gg^{D} v_0\,;\mu_0$ for some $v_0 \in \mathsf{V}$. We say $e\,;\mu$ *evaluates* to $v_0\,;\mu_0$ with respect to D when $e\,;\mu \gg^{D} v_0\,;\mu_0$ for some $v_0 \in \mathsf{V}$. More often than not we leave D implicit and simply write $e\,;\mu \gg v_0\,;\mu_0$. We can now easily describe the functions determined by our definition D. Namely if ϑ is defined in D and $y_1, \ldots, y_n$ are its arguments then the corresponding partial function $\vartheta^{D}\colon \mathsf{M}^{(n)}_{sexp} \rightsquigarrow \mathsf{M}_{sexp}$ is defined by $\vartheta^{D}([v_1, \ldots, v_n]\,;\mu) = v_0\,;\mu_0$ iff $\vartheta(v_1, \ldots, v_n)\,;\mu \gg^{D} v_0\,;\mu_0$.

It is easy to see that for any memory object description, at most one of the single step rules applies. Thus the single step relation is functional as is the corresponding evaluation relation $e\,;\mu_0 \gg^{D} v\,;\mu_1$.

We use memory operation and function symbols in two contexts: in terms denoting memory objects and in memory object descriptions. In the term context we include the memory as an argument while in the memory object description the memory is not included in the argument. For example, $car(c\,;\mu)$ is a term and $car(c)\,;\mu$ is a memory object description, and we have $car(c)\,;\mu \gg^{D} car(c\,;\mu)$. The two uses of operation and function symbols should cause no confusion.

4. Equivalence Relations

Equivalence relations on expressions and terms are central to our approach. Operations on programs need meanings to transform and meanings to preserve, (Talcott, 1986), and the study of various notions of equivalence is simply a study of the various *meanings*. As we have mentioned before, we regard verification of programs and derivation of programs as duals of one another. In verification one proves that a given program meets or satisfies a certain specification. In derivation one does the reverse — from specifications one derives programs which accomplish the task. Often such specifications are just simple-minded or well known programs that perform the required calculation or construction. When this is the case, satisfying a specification or accomplishing a task can be

formulated and proved using these equivalence relations. Without exception these equivalence relations are generated by equivalence relations on memory object descriptions. There are two different types of equivalence relations one can define on memory object descriptions and terms. First are the *extensional* ones, which are generated by equivalence relations on memory objects and are thus really properties of the function or transformation denoted by the expression. Second are the *intensional* ones, which not only depend on value of the memory object description but how the memory has been transformed in the process of evaluating it. The latter class turns out to have a much more manageable theory than the former. We give two examples of the former, namely isomorphism and Lisp equality, and one of the latter, namely strong isomorphism. It will become apparent to the reader that strong isomorphism stands to destructive Lisp as Lisp equality stands to pure Lisp; in fact we shall state a theorem to that effect.

4.1. Isomorphism and Lisp Equality

In this section we describe two extensional equivalence relations, Lisp equality and isomorphism. Both of these are equivalence relations on memory objects, memory object descriptions and expressions, and are generated in a uniform fashion from their behavior on memory objects. Thus we describe in general how to extend an equivalence relation on memory objects to one that includes memory object descriptions and expressions, consequently we need only define Lisp equality and isomorphism on memory objects.

Supposing $\sim$ is an equivalence relation on memory objects, we extend $\sim$ to memory object descriptions and expressions in the following fashion.

1. **Memory object descriptions:** Two memory object descriptions $e_0 ; \mu_0$ and $e_1 ; \mu_1$ are said to be $\sim$-related, again written $e_0 ; \mu_0 \sim e_1 ; \mu_1$, iff either they both fail to denote or else they denote $\sim$-related memory objects.

2. **Expressions:** Two expressions $e_0(\bar{x})$ and $e_1(\bar{x})$ are $\sim$-related, $e_0(\bar{x}) \sim e_1(\bar{x})$, iff $\forall\ \bar{v} ; \mu \in \mathsf{M}_{sexp}$ we have that $e_0(\bar{v}) ; \mu \sim e_1(\bar{v}) ; \mu$.

The actual algebraic definition of these relations on memory objects uses the path function $\partial_{v;\mu}$ associated with $v;\mu$. We say memory objects $\bar{v}_0;\mu_0$ and $\bar{v}_1;\mu_1$ are isomorphic if and only if they are graph theoretically isomorphic via a map which preserves atomic values. This is stated precisely in the following.

Definition: If $[v_0,\ldots,v_n] ; \mu$, $[v'_0,\ldots,v'_n] ; \mu' \in \mathsf{M}^{(n+1)}_{sexp}$ we say $[v_0,\ldots,v_n] ; \mu$ is *isomorphic* to $[v'_0,\ldots,v'_n];\mu'$, written $[v_0,\ldots,v_n];\mu \cong [v'_0,\ldots,v'_n];\mu'$, if there is a bijection $h : \mathsf{V} \to \mathsf{V}$ which is the identity on A, maps $\mathsf{C} \to \mathsf{C}$ and is such that $h \circ \partial_{v_i;\mu} = \partial_{v'_i;\mu'}$ as partial functions, for every $i \in n+1$ (here $\circ$ denotes function composition). It is sometimes convenient to specify the function h, to do this we write $\bar{v} ; \mu \cong \bar{v}' ; \mu'$ via h.

The fact that isomorphism between terms can be axiomatized is expressed by the following result.

Theorem: **(Effectiveness of $\cong$)** There is a decision procedure which determines whether or not two primitive terms are isomorphic.

Remark: This theorem does not hold for arbitrary expressions, simply because ϑ is total iff $\mathtt{if}(\vartheta(x),\mathtt{T},\mathtt{T}) \cong \mathtt{T}$, where ϑ is any defined function.

Two memory objects $\bar{v}_0 ; \mu_0$ and $\bar{v}_1 ; \mu_1$ are said to be *Lisp equal* iff they have exactly the same *car-cdr* chains, and whenever one chain ends in an atom in one it ends in the same atom in the other. This is formulated precisely by:

Definition: We say $v_0;\mu_0$ and $v_1;\mu_1$ are *Lisp equal*, $v_0;\mu_0 \equiv v_1;\mu_1$, iff $\partial_{v_0;\mu_0}$ and $\partial_{v_1;\mu_1}$ have the same domains and $\partial_{v_0;\mu_0}(\sigma) = a$ iff $\partial_{v_1;\mu_1}(\sigma) = a$, for $\sigma \in \delta_{\partial_{v_i;\mu_i}}$, $a \in \mathbb{A}$. Finally we say $[v_0, \ldots, v_n] ; \mu \equiv [v'_0, \ldots, v'_n] ; \mu'$ if for each $0 \leq i \leq n$ we have that $v_i ; \mu \equiv v'_i ; \mu'$.

Notice that $v_0 ; \mu_0 \equiv v_1 ; \mu_1$ means that $v_0 ; \mu_0$ and $v_1 ; \mu_1$ *print* the same (for typical Lisp printing algorithms). An important point to observe is that S-expression memory operations preserve isomorphism. For example, $[c, v] ; \mu \cong [c', v'] ; \mu'$ implies that $rplaca([c, v] ; \mu) \cong rplaca([c', v'] ; \mu')$. This is certainly not true for Lisp equality as the first example below indicates. Note that isomorphism and Lisp equality also differ in that

$$\bigwedge_{0 \leq i \leq n} v_i ; \mu \sim v'_i ; \mu' \rightarrow [v_0, \ldots, v_n] ; \mu \sim [v'_0, \ldots, v'_n] ; \mu'$$

is true when $\sim$ is $\equiv$ but not when it is $\cong$. The former unlike the latter does not take structure sharing into account. The following are simple examples of these two relations.

Examples:

1. $cons(cons(\mathrm{T}, \mathrm{T}), cons(\mathrm{T}, \mathrm{T})) \equiv \mathtt{let}\{x \leftarrow cons(\mathrm{T}, \mathrm{T})\}\ cons(x, x)$. Note that we clearly have $cons(cons(\mathrm{T}, \mathrm{T}), cons(\mathrm{T}, \mathrm{T})) \not\cong \mathtt{let}\{x \leftarrow cons(\mathrm{T}, \mathrm{T})\}\ cons(x, x)$, and that the expression $eq(car(y), cdr(y))$ will return NIL when evaluated on the object denoted by the left hand side and return T for the right hand side.

2. $\mathtt{if}(atom(x), \mathrm{T}, rplacd(rplaca(x, \mathrm{T}), \mathrm{T})) \cong \mathtt{if}(atom(x), \mathrm{T}, cons(\mathrm{T}, \mathrm{T}))$.

Note these examples show that $e_0 \sim e_1$ does not imply that $e(\bar{x}, e_0) \sim e(\bar{x}, e_1)$ when $\sim$ is either $\cong$ or $\equiv$. In the case of the first example $e(y) = rplaca(car(y), \mathrm{NIL})$ suffices and in the second case $e(x, y) = rplaca(y, x)$ is sufficient. Thus we do not have Leibniz's Law for these two relations. Consequently to prove properties of expressions using these two relations one must explicitly carry the memory around, dissecting it when necessary. In other words to prove that certain expressions are related, one must work at the next level down, the level of memory object descriptions. This is what we mean by *hand simulation*, in the *transformation paradigm* one need not descend to this level. As we have already mentioned we have the following simple result:

Proposition:

1. $\equiv$ and $\cong$ are both equivalence relations, on memory objects, memory object descriptions and expressions.

2. If $v_0 ; \mu_0 \cong v_1 ; \mu_1$ then $v_0 ; \mu_0 \equiv v_1 ; \mu_1$. The converse is false.

3. If D is a definition in either M_{sexp}, M_{pure^+} or M_{pure} and ϑ is a function defined in D then the partial function determined by this definition preserves isomorphism. By this we mean that if $[v_0, \ldots, v_n];\mu \cong [v'_0, \ldots, v'_n];\mu'$ then $\vartheta(v_0, \ldots, v_n);\mu \cong \vartheta(v'_0, \ldots, v'_n);\mu'$ whenever either (equivalently both) denote.

The third clause in this proposition would be false if we did not abide by our convention that expressions do not contain cells. Thus providing us with another justification for

our convention. These properties in a sense characterize isomorphism. It is the weakest equivalence relation generated from an equivalence relation on memory objects that satisfies this proposition. For consider the following result, the proof of which is left to the reader.

Theorem: Suppose that $\sim$ is an equivalence relation on memory objects, memory object descriptions and expressions such that

1. $e_0(\bar{v}_0);\mu_0 \sim e_1(\bar{v}_0);\mu_0$ implies that either both sides fail to denote, or else they denote memory objects which are $\sim$-related.
2. $(\forall \bar{v}\,;\mu \in \mathsf{M}_{sexp})(e_0(\bar{v})\,;\mu \sim e_1(\bar{v})\,;\mu)$ iff $e_0(\bar{x}) \sim e_1(\bar{x})$.
3. The relation $\sim$ satisfies a very weak Leibniz's Law

$$\bar{v}_0\,;\mu_0 \sim \bar{v}_1\,;\mu_1 \wedge e_0(\bar{x}) \sim e_1(\bar{x}) \rightarrow e_0(\bar{v}_0)\,;\mu_0 \sim e_1(\bar{v}_1)\,;\mu_1.$$

Then $e_0 \sim e_1 \rightarrow e_0 \cong e_1$.

A similar characterization of Lisp equality is also possible using the M_{pure} fragment.

Theorem: Restricting ourselves to M_{pure}, suppose that $\sim$ is an equivalence relation on memory objects, memory object descriptions and expressions such that:

1. $e_0(\bar{v}_0);\mu_0 \sim e_1(\bar{v}_0);\mu_0$ implies that either both sides fail to denote, or else they denote memory objects which are $\sim$-related.
2. $(\forall \bar{v}\,;\mu \in \mathsf{M}_{pure})(e_0(\bar{v})\,;\mu \sim e_1(\bar{v})\,;\mu)$ iff $e_0(\bar{x}) \sim e_1(\bar{x})$.
3. The relation $\sim$ satisfies Leibniz's Law: Supposing that $e_0(\bar{x}), e_1(\bar{x}), e(\bar{x}, y)$ are in M_{pure} then $e_0(\bar{x}) \sim e_1(\bar{x}) \rightarrow e(\bar{x}, e_0(\bar{x})) \sim e(\bar{x}, e_1(\bar{x}))$.

Then $e_0 \sim e_1 \rightarrow e_0 \equiv e_1$.

The following result is slightly stronger than the corresponding result for pure Lisp; this is because in the following we must take structure sharing and cyclicity into account.

Theorem: (Effectiveness of $\equiv$) There is a decision procedure which determines whether or not two primitive terms are Lisp equal. As in the case of isomorphism this result does not hold for expressions.

4.2. Strong Isomorphism

These extensional relations are not strong enough for many purposes. For example in program transformations and the like one is not just concerned with the value of an expression but also with the effect it has on memory. It is also desirable to have a relation between expressions that satisfies Leibniz's Law, allowing syntactical manipulations in verifications and derivations. In other words a relation which lends itself to the transformation plus induction paradigm. For this we define the stronger notion of two terms $e_0(\bar{x})$ and $e_1(\bar{x})$ being strongly isomorphic, written $e_0(\bar{x}) \simeq e_1(\bar{x})$, which is true if they are not only isomorphic but also they make, up to this isomorphism and the production of garbage, exactly the same modifications to the memory they are evaluated in. By garbage

we mean cells constructed in the process of evaluation that are not accessible from either the result or the, possibly modified, arguments. This entails that the isomorphism must map newly created cells to newly created cells and be the identity on old ones. The relation is then extended to expressions in the same way as in the extensional case. As an example neither of the two examples of isomorphic and Lisp equal expressions given previously are strongly isomorphic. This relation is used in specifying and constructing program transformations and we shall define it and study its elementary properties in this and subsequent sections.

Definition: Two memory object descriptions $e_0(\bar{v})\ ;\ \mu$ and $e_1(\bar{v})\ ;\ \mu$ are said to be *strongly isomorphic*, written $e_0(\bar{v})\ ;\ \mu \simeq e_1(\bar{v})\ ;\ \mu$, iff either they both fail to denote or else they both denote, and for $w_i\ ;\ \mu_i$ with $e_i(\bar{v})\ ;\ \mu \gg w_i\ ;\ \mu_i$ for $i \in 2$, there is a bijection $h : \mathsf{V} \to \mathsf{V}$ which is the identity on $\mathsf{A} \cup \mathbf{Cells}_\mu(\bar{v})$ and maps $\mathsf{C} \to \mathsf{C}$ so that the following two properties hold: $w_0\ ;\ \mu_0 \cong w_1\ ;\ \mu_1$ via h; $\mathbf{Cells}_\mu(\bar{v})\ ;\ \mu_0 \cong \mathbf{Cells}_\mu(\bar{v})\ ;\ \mu_1$ via h.

Remarks:

1. Note that the notion of strong isomorphism is *weaker* than saying that the descriptions denote isomorphic objects (when either denotes) and that their evaluation modifies memory in exactly the same way.
2. In the second condition we are regarding $\mathbf{Cells}_\mu(\bar{v})$ as a sequence, ordered in some fixed fashion.
3. It follows from the second condition, by a simple pasting argument and the fact that μ_0 must agree with μ on $\delta_\mu - \mathbf{Cells}_\mu(\bar{v})$, that we can actually require $\delta_\mu\ ;\ \mu_0 \cong \delta_\mu\ ;\ \mu_1$ via such an h.
4. The extension of this definition to the M_{mfsexp} case is the obvious one and so we leave it as an exercise for the so-inclined reader.

Definition: Two expressions $e_0(\bar{x})$ and $e_1(\bar{x})$ are said to be *strongly isomorphic*, written $e_0(\bar{x}) \simeq e_1(\bar{x})$, iff $(\forall \bar{v}\ ;\ \mu \in \mathsf{M}_{sexp})\ \ (e_0(\bar{v})\ ;\ \mu \simeq e_1(\bar{v})\ ;\ \mu)$.

A simple example of a transformation which preserves strong isomorphism is that of unfolding, named after the unfold transformation in (Burstall and Darlington, 1977). The *unfolding*, $e^{\bowtie}$, of an expression e with respect to the definition D is defined by:

$$e^{\bowtie} = \begin{cases} e & \text{if } e \in \mathsf{X} \cup \mathsf{V}, \\ \mathtt{if}(e^{\bowtie}_{test}, e^{\bowtie}_{then}, e^{\bowtie}_{else}) & \text{if } e = \mathtt{if}(e_{test}, e_{then}, e_{else}), \\ \mathtt{seq}(e^{\bowtie}_0, \ldots e^{\bowtie}_n) & \text{if } e = \mathtt{seq}(e_0, \ldots e_n), \\ \mathtt{let}_{0 \le i \le n}\{y_i \leftarrow e^{\bowtie}_i\} e^{\bowtie}_{body} & \text{if } e = \mathtt{let}_{0 \le i \le n}\{y_i \leftarrow e_i\} e_{body}, \\ \vartheta(e^{\bowtie}_1, \ldots, e^{\bowtie}_n) & \text{if } e = \vartheta(e_1, \ldots, e_n) \text{ and } \vartheta \in \mathsf{O}_{sexp}, \\ \mathtt{let}_{0 \le i \le n}\{y_i \leftarrow e^{\bowtie}_i\} e_\vartheta & \text{if } e = \vartheta(e_1, \ldots, e_n) \text{ and } \vartheta(\bar{y}) \leftarrow e_\vartheta \in D, \\ \vartheta(e^{\bowtie}_1, \ldots, e^{\bowtie}_n) & \text{otherwise .} \end{cases}$$

Unfolding Theorem: For any expression $e(\bar{x})$, $e(\bar{x}) \simeq e^{\bowtie}(\bar{x})$.

Proposition: $\simeq$ is an equivalence relation on both memory object descriptions and expressions.

Theorem: (Effectiveness of $\simeq$) There is a decision procedure which determines whether or not two terms are strongly isomorphic.

Remarks: Yet again the theorem does not hold for arbitrary expressions, simply because $\mathtt{if}(\vartheta(x), \mathtt{T}, \mathtt{T}) \simeq \mathtt{T}$ holds iff ϑ is total, where ϑ is any purely defined function.

The main motivation for studying this relation is given by the following theorem. It and its corollaries describe fundamental syntactic manipulations that preserve strong isomorphism. In particular they have the consequence that Leibniz's Law holds for strong isomorphism.

Substitution Theorem: If $e^0_{body}(\bar{x}, \bar{y}) \simeq e^1_{body}(\bar{x}, \bar{y})$, $|\bar{x}| = k+1$ and $e^0_i(\bar{y}) \simeq e^1_i(\bar{y})$, for $0 \leq i \leq k$ then $\mathtt{let}_{0 \leq i \leq k}\{x_i \leftarrow e^0_i(\bar{y})\}e^0_{body}(\bar{x}, \bar{y}) \simeq \mathtt{let}_{0 \leq i \leq k}\{x_i \leftarrow e^1_i(\bar{y})\}e^1_{body}(\bar{x}, \bar{y})$.

Remark: Notice that while $eq(x, x) \simeq \mathtt{T}$ we have $eq(cons(y, y), cons(y, y)) \simeq \mathtt{NIL}$ which is why we have formulated the theorem in the above fashion.

The following corollaries follow with a minimum of effort and are left as exercises.

Corollary 1: (Sequencing theorem) If $e^0_i(\bar{x}) \simeq e^1_i(\bar{x})$, for $0 \leq i < k$ then

$$\mathtt{seq}_{0 \leq i < k}(e^0_i(\bar{y})) \simeq \mathtt{seq}_{0 \leq i < k}(e^1_i(\bar{y})).$$

Corollary 2: (Composition theorem) If $e^0_i(\bar{x}) \simeq e^1_i(\bar{x})$, for $0 \leq i < k$ and ϑ is either an n-ary memory operation or an n-ary function symbol then

$$\vartheta(e^0_0(\bar{y}), \ldots, e^0_{k-1}(\bar{y})) \simeq \vartheta(e^1_0(\bar{y}), \ldots, e^1_{k-1}(\bar{y})).$$

Corollary 3: (Branching theorem) If $e^0_i(\bar{x}) \simeq e^1_i(\bar{x})$, for $0 \leq i < 3$ then

$$\mathtt{if}(e^0_0(\bar{y}), e^0_1(\bar{y}), e^0_2(\bar{y})) \simeq \mathtt{if}(e^1_0(\bar{y}), e^1_1(\bar{y}), e^1_2(\bar{y})).$$

The most useful consequence of the Substitution Theorem is Leibniz's Law; replacing a subexpression of an expression by a strongly isomorphic one preserves strong isomorphism:

Theorem: (Leibniz's Law) Supposing $e_i(\bar{x}), e(\bar{x}, y)$ are expressions, $i < 2$. Then, $e_0(\bar{x}) \simeq e_1(\bar{x}) \rightarrow e(\bar{x}, e_0(\bar{x})) \simeq e(\bar{x}, e_1(\bar{x}))$.

It is probably an appropriate time to point out the following property of strong isomorphism: it is the weakest relation extending Lisp equality that satisfies Leibniz's Law.

Theorem: Suppose that $\sim$ is an equivalence relation on memory object descriptions and expressions such that:

1. $(\forall \bar{v}\,;\mu \in \mathsf{M}_{sexp})(e_0(\bar{v})\,;\mu \sim e_1(\bar{v})\,;\mu)$ iff $e_0(\bar{x}) \sim e_1(\bar{x})$.
2. $e_0(\bar{v})\,;\mu \sim e_1(\bar{v})\,;\mu$ implies that $e_0(\bar{v})\,;\mu \equiv e_1(\bar{v})\,;\mu$.
3. If $e_0(\bar{x}), e_1(\bar{x}), e(\bar{x}, y) \in \mathsf{E}$, then $e_0(\bar{x}) \sim e_1(\bar{x}) \rightarrow e(\bar{x}, e_0(\bar{x})) \sim e(\bar{x}, e_1(\bar{x}))$.

Then $e_0 \sim e_1 \rightarrow e_0 \simeq e_1$.

We finish with two definitions that will be used heavily in the sequel. The two notions to be defined are that of a *gentle* expression, being one that does not alter any memory it is evaluated in, and that of a expression being a *principal* subexpression of a given expression. By the latter is meant a subexpression which must be evaluated first, prior to evaluating the whole. If z is principal in $e(z)$ then the first expression to be evaluated in $e(e_0)$ is e_0. We call e_0 in such a situation a *principal subexpression* of e.

Definition: $e(\bar{x}) \in \mathsf{E}$ is *gentle* iff for every $i < |\bar{x}|$, $\mathtt{seq}(e(\bar{x}), \bar{x}{\downarrow_i}) \simeq \bar{x}{\downarrow_i}$.

Definition: We say that the variable z is *principal* in the expression e if and only if

$$e = \begin{cases} z & \text{or,} \\ \mathtt{if}(e', e_1, e_2) & \text{or,} \\ \mathtt{if}(\chi_1, e_1', e_2') & \text{or,} \\ \vartheta(\chi_1, \ldots, \chi_n, e', e_1, \ldots, e_m) & \text{or,} \\ \mathtt{seq}(\chi_1, \ldots, \chi_n, e', e_1, \ldots, e_m) & \text{or,} \\ \mathtt{let}\{x_1 \leftarrow \chi_1, \ldots x_n \leftarrow \chi_n\}e' & \text{or,} \\ \mathtt{let}_{\substack{1 \le i \le n \\ 1 \le j < m}} \{x_i \leftarrow \chi_i, x_{n+1} \leftarrow e', x_{n+1+j} \leftarrow e_j\}e_m & \end{cases}$$

assumming that $z \in \mathsf{X}$, $\chi_1, \ldots \chi_n \in \mathsf{V} \cup (\mathsf{X} - \{z\})$, $x_1, \ldots, x_{m+n+1} \in \mathsf{X} - \{z\}$ are distinct, $e_1, \ldots e_m \in \mathsf{E}$ do not have z as a free variable, and z is principal in e' and e_i' for $i < 2$.

The proof of the following theorem is a simple induction on the rank of e.

Principal Subexpression Theorem: If z is principal in $e(z)$ then

1. $\mathtt{seq}(e_0, \ldots, e_{n-1}, e(e_n)) \simeq e(\mathtt{seq}(e_0, \ldots, e_{n-1}, e_n))$,
2. $\mathtt{let}\{x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n, z \leftarrow e_0\}e(z) \simeq \mathtt{let}\{x_1 \leftarrow e_1, \ldots x_n \leftarrow e_n\}e(e_0)$, provided that e_0 contains no free variables from $\bar{x}$.
3. $e(\mathtt{if}(e_0, e_1, e_2)) \simeq \mathtt{if}(e_0, e(e_1), e(e_2))$.
4. If $e(z) \simeq \mathtt{seq}(e(z), z)$ then $\mathtt{let}\{z \leftarrow e_0\}\mathtt{seq}(e(z), e_1, \ldots) \simeq \mathtt{let}\{z \leftarrow e(e_0)\}\mathtt{seq}(e_1, \ldots)$.

5. Axioms for $\simeq$

In this section we shall present a large collection of principles, which for want of a better word, we shall call *axioms*. We make no claim as to their *independence*. Nor, unfortunately, shall we claim *completeness*. Firstly however we provide the reader with a perspective. We have divided the axioms relating $\simeq$ into groups, depending upon the *principal* entity that the property depends upon. The branching construct, `if`, and its properties are well understood, see for example (Bloom and Tindell, 1983) or (Guessarian and Meseguer, 1985) and so we shall not dwell upon them here.

5.1. The Data Operations

The properties of the data operations are quite simple to express and we expect the following list to be fairly close to being complete (we shall point out places where there is room for improvement). We begin by enumerating the properties of *rplaca* and *rplacd*,

necessitating heavy use of seq. The first simply states that $rplaca(x,y)$ and $rplacd(x,y)$ both return the modified x.

1. $\mathrm{seq}(rplaca(x,y),x) \simeq rplaca(x,y)$ and $\mathrm{seq}(rplacd(x,y),x) \simeq rplacd(x,y)$.

The next describes the modifications made to x, and the two after describe what isn't altered.

2. $car(rplaca(x,y)) \simeq \mathrm{seq}(rplaca(x,y),y)$ and $cdr(rplacd(x,y)) \simeq \mathrm{seq}(rplacd(x,y),y)$.
3. $car(rplacd(x,y)) \simeq \mathtt{let}\{z \leftarrow car(x)\}\mathrm{seq}(rplacd(x,y),z)$.
4. $cdr(rplaca(x,y)) \simeq \mathtt{let}\{z \leftarrow cdr(x)\}\mathrm{seq}(rplaca(x,y),z)$.

Notice that in these last two the let is important since simply saying

$$car(rplacd(x,y)) \simeq \mathrm{seq}(rplacd(x,y), car(x)),$$

for example, would not be enough. In fact this last equation would follow from 1. using the principal subexpression theorem. These last two can also be generalized so that they imply that every other pointer in existing cells, other than the one modified, remains the same. We shall not do this.

The next four axioms are *cancellation* axioms. The first two simply state that modifying the same pointer in x successively is identical to carrying out the single second modification. The third states simple relations between *cons*-ing a new cell and modifying a newly constructed one. The final one just expresses the fact that altering a pointer so that it now points to what it did previously is no alteration.

5. $\mathrm{seq}(rplaca(x,y_0), rplaca(x,y_1)) \simeq rplaca(x,y_1)$.
6. $\mathrm{seq}(rplacd(x,y_0), rplacd(x,y_1)) \simeq rplacd(x,y_1)$.
7. $rplaca(cons(z,y),x) \simeq cons(x,y)$ and $rplacd(cons(x,z),y) \simeq cons(x,y)$.
8. $rplaca(x, car(x)) \simeq x$ and $rplacd(x, cdr(x)) \simeq x$ when $x \in \mathbb{C}$.

The next three properties of *rplaca* and *rplacd* are simple *commutativity* properties, stating when the order of these destructive modifications can be changed without any harm.

9. $\mathrm{seq}(rplacd(x_0,x_1), rplaca(x_2,x_3), x_4) \simeq \mathrm{seq}(rplaca(x_2,x_3), rplacd(x_0,x_1), x_4)$.
10. $\mathrm{seq}(rplaca(x_0,x_1), rplaca(x_2,x_3), x_4) \simeq \mathrm{seq}(rplaca(x_2,x_3), rplaca(x_0,x_1), x_4)$, if $x_0 \neq x_2$.
11. $\mathrm{seq}(rplacd(x_0,x_1), rplacd(x_2,x_3), x_4) \simeq \mathrm{seq}(rplacd(x_2,x_3), rplacd(x_0,x_1), x_4)$, if $x_0 \neq x_2$.

The next two imply that one cannot modify an atom.

12. $atom(rplaca(e_0,e_1)) \simeq \mathrm{seq}(rplaca(e_0,e_1), \mathrm{NIL})$.
13. $atom(rplacd(e_0,e_1)) \simeq \mathrm{seq}(rplacd(e_0,e_1), \mathrm{NIL})$.

What remains now are simple properties of the remaining operations. The first simply states that *cons* creates a cell, never an atom. The next two are simple facts about the characteristic function of A, *atom*.

14. $atom(cons(x,y)) \simeq \mathtt{NIL}$.

15. $atom(x) \simeq \mathtt{T}$ or $atom(x) \simeq \mathtt{NIL}$.

16. $atom(\mathtt{T}) \simeq \mathtt{T}$ and $atom(\mathtt{NIL}) \simeq \mathtt{T}$.

The following properties relate the cell constructing operation, *cons*, with the access operations, *car* and *cdr*. Note that $cons(car(x), cdr(x)) \not\simeq x$ regardless of the nature of x.

17. $car(cons(x,y)) \simeq x$ and $cdr(cons(x,y)) \simeq y$.

The next axiom states that *cons* always creates a new cell. The remaining five axioms state simple properties of the identity, *eq*. There are obvious modifications of these axioms for the case of *atom*:*eq*. We leave their statement to the reader's imagination. There are also axioms concerning the partial nature of the operations which we do not include here.

18. $eq(cons(e_0,e_1),e_2) \simeq \mathtt{seq}(e_0,e_1,e_2,\mathtt{NIL})$.

19. $eq(e_0,e_1) \simeq eq(e_1,e_0)$ if the e_i are gentle.

20. $eq(x,y) \simeq \mathtt{NIL}$ if $x \neq y$, and $eq(x,y) \simeq \mathtt{T}$ if $x = y$.

21. $eq(\mathtt{T},\mathtt{NIL}) \simeq \mathtt{NIL}$.

22. $atom(eq(e_0,e_1)) \simeq \mathtt{seq}(e_0,e_1,\mathtt{T})$.

5.2. Properties of seq

We now commence with the control structure, beginning with seq. The first three properties are the simplest one can think of concerning the *sequencing* primitive: a sequence of one expression is the same as the expression; a sequence within a sequence can be flattened, i.e. nested seq removal; and thirdly, seq can be defined in terms of let.

1. $\mathtt{seq}(e) \simeq e$

2. $\mathtt{seq}(e_0,\ldots,\mathtt{seq}(e_n\ldots e_m),e_k,\ldots) \simeq \mathtt{seq}(e_0,\ldots,e,e_n,\ldots e_m,e_k,\ldots)$

3. $\mathtt{seq}(e_0,e_1,\ldots,e_n) \simeq \mathtt{let}\{x_0 \leftarrow e_0, x_1 \leftarrow e_1,\ldots,x_n \leftarrow e_n\}x_n$

The next six properties all stem from the fact that the value returned by the sequence is the value of the last expression in the sequence. Consequently gentle expressions, when they are defined, can be added and removed from anywhere within the sequence other than the last. The five axioms subsequent to the first allow one to replace an expression in a value-ignoring position by one or more that have the same effect on memory. Note that in the last two of the six, the hypothesis cannot be weakened to $atom(e_n) \simeq \mathtt{seq}(e_n,\mathtt{NIL})$ for reasons of validity.

4. $\mathtt{seq}(e_0(\bar{y})\ldots,e_n(\bar{y}),e(\bar{y})) \simeq e(\bar{y})$ if $e_i(\bar{y})$ are total gentle terms, $0 \leq i \leq n$.

5. $\mathtt{seq}(e_0,\ldots,cons(e_n,e_{n+1}),\ldots,e_m) \simeq \mathtt{seq}(e_0,\ldots,e_n,e_{n+1},\ldots,e_m)$.

6. $\mathtt{seq}(e_0,\ldots,eq(e_n,e_{n+1}),\ldots,e_m) \simeq \mathtt{seq}(e_0,\ldots,e_n,e_{n+1},\ldots,e_m)$.

7. $\mathtt{seq}(e_0, \ldots, atom(e_n), \ldots, e_m) \simeq \mathtt{seq}(e_0, \ldots, e_n, \ldots, e_m)$.
8. If $atom(\mathtt{seq}_{0 \le i \le n}(e_i)) \simeq \mathtt{seq}_{0 \le i \le n}(e_i, \mathrm{NIL})$, then

$$\mathtt{seq}_{\substack{0 \le i < n \\ n < j \le m}}(e_i, car(e_n), e_j) \simeq \mathtt{seq}_{0 \le i \le m}(e_i).$$

9. If $atom(\mathtt{seq}_{0 \le i \le n}(e_i)) \simeq \mathtt{seq}_{0 \le i \le n}(e_i, \mathrm{NIL})$, then

$$\mathtt{seq}_{\substack{0 \le i < n \\ n < j \le m}}(e_i, cdr(e_n), e_j) \simeq \mathtt{seq}_{0 \le i \le m}(e_i).$$

A shortcoming in our axioms regarding seq is in their dealing with the problem of commuting expressions. An elegant description of when such a principle is valid has eluded us.

5.3. Properties of let

The lexically scoped variable binding operation, let, is by far the most difficult of the control primitives to describe axiomatically. Even though we show it is possible in (Mason, 1986), we have but scratched the surface here. The first property is a simple consequence of the *lexical* nature of let.

1. $\mathtt{let}\{x_0 \leftarrow e_0\}\mathtt{let}\{x_1 \leftarrow e_1\}e \simeq \mathtt{let}\{x_0 \leftarrow e_0, x_1 \leftarrow e_1\}e$, if $x_0 \notin FV(e_1)$.

The next two properties are the simplest forms of let elimination we have found. They are by no means the end of the story. There are many other cases when one can eliminate or introduce a let but we have not found any elegant way of describing them. Basically there are only two reasons why a let cannot be eliminated. The first is when it is used to obtain a pointer to an object about to become inaccessible. For example

$$\mathtt{let}\{y \leftarrow car(x)\}\mathtt{seq}(rplaca(x, z), \ldots).$$

The second is when it is used to obtain a pointer to the value of an expression e that when evaluated twice produces distinct answers. For example

$$\mathtt{let}\{y \leftarrow cons(\mathrm{T}, \mathrm{T})\}\ cons(y, y).$$

We would be delighted if someone produced an elegant characterization of when a let can be eliminated. The list of properties of let continues as follows.

2. $\mathtt{let}_{\substack{0 \le i \le n \\ 0 \le j \le m}}\{x_i \leftarrow e_i, y_j \leftarrow e'_j\}e \simeq \mathtt{seq}_{0 \le i \le n}(e_i, \mathtt{let}_{0 \le j \le m}\{y_j \leftarrow e'_j\}e), x_i \notin FV(e), i \le n$.
3. $\mathtt{let}\{y_0 \leftarrow e_0, \ldots, x \leftarrow v, \ldots y_n \leftarrow e_n\}e(x, \bar{y}) \simeq \mathtt{let}_{0 \le i \le n}\{y_i \leftarrow e_i\}e(v, \bar{y})$.

The last axiom gives a sufficient condition for commuting binding expressions. It is clear that this condition is not necessary, but again an elegant description of when such principles are valid has escaped us.

4. If $e_i, i < 2$, are gentle, then $\mathtt{let}\{x_0 \leftarrow e_0, x_1 \leftarrow e_1\}e \simeq \mathtt{let}\{x_1 \leftarrow e_1, x_0 \leftarrow e_0\}e$.

Further instances of valid let elimination can be found in (Mason, 1986).

6. Three Examples

In this section we give four simple applications of the theory. Each illustrates a particular technique or style. A plethora of applications can be found in (Mason, 1986).

6.1. Example 1: The *defined:eq* Program

We chose this as our first real example of the use of strong isomorphism for four reasons. Firstly it is a very simple program, involving no recursion, that accomplishes an easily specified task. The second reason is that the correctness proof is quite subtle, and involves several delicate manipulations. Thirdly, each step in the proof is a simple combination of the axioms in the preceding section. Finally it demonstrates that there are only three natural fragments of destructive Lisp, in that identity of cells is intimately connected with mutability . We shall refer to the axioms in a systematic and self-explanatory fashion. For example do4. refers to the fourth axiom concerning the data operations, while seq1. refers to the first property of seq and PST refers to the principal subexpression theorem. We begin, by force, with the definition of the *defined:eq*.

```
defined:eq(x,y) ←
    if(or(atom(x), atom(y)),
      atom:eq(x,y),
      let{xa ← car(x), ya ← car(y)}
        seq(rplaca(x,T),
            rplaca(y,NIL),
            let{ans ← atom:eq(car(x), car(y))}
              seq(rplaca(x,xa), rplaca(y,ya), ans))).
```

The theorem that we prove is, predictably, the following:

Theorem: $defined{:}eq(x,y) \simeq eq(x,y)$.

Proof of Theorem: Clearly the result will be true when either x or y is an atom. Consequently we need only consider the case when they are both cells. We can simplify the problem even further by observing that the theorem is equivalent to the following two statements: (1.) $defined{:}eq(x,y) \simeq \mathtt{NIL}$ whenever $x \neq y$; (2.) $defined{:}eq(x,y) \simeq \mathtt{T}$ whenever $x = y$. We prove the first of these, the second is a simple variation.

Proof of 1: Suppose that both x and y are distinct cells with $\mathrm{x_a}$ and $\mathrm{y_a}$ their respective cars. Evaluating the body of *defined:eq*, simplifying the if and eliminating the let in this context results in:

```
seq(rplaca(x,T),
    rplaca(y,NIL),
    let{ans ← atom:eq(car(x), car(y))}seq(rplaca(x,xa), rplaca(y,ya), ans))
```

It is this expression that, we shall demonstrate, is strongly isomorphic to NIL.

$$\begin{aligned}
&\mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\\
&\quad \mathit{rplaca}(y,\mathrm{NIL}),\\
&\quad \mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathit{car}(x),\mathit{car}(y))\}\mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans})).
\end{aligned}$$

Pushing the seq inside the let produces (PST)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{atom{:}eq}(\mathit{car}(x),\mathit{car}(y))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

And now pushing the seq inside the *atom:eq* yields (PST)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{car}(x)),\mathit{car}(y))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Commuting the two calls to *rplaca* gives (do10, seq2)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathtt{seq}(\mathit{rplaca}(y,\mathrm{NIL}),\mathit{rplaca}(x,\mathrm{T}),\mathit{car}(x)),\mathit{car}(y))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Now eliminating the call to *car*, (PST, do2, do1, seq2)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathtt{seq}(\mathit{rplaca}(y,\mathrm{NIL}),\mathit{rplaca}(x,\mathrm{T}),\mathrm{T}),\mathit{car}(y))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Popping the seq out of the *atom:eq* results in (PST)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathtt{seq}(\mathit{rplaca}(y,\mathrm{NIL}),\mathit{rplaca}(x,\mathrm{T}),\mathit{atom{:}eq}(\mathrm{T},\mathit{car}(y)))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Commuting the calls to *rplaca* has the consequence that (do10, seq2)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{atom{:}eq}(\mathrm{T},\mathit{car}(y)))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Now commuting the *gentle* arguments to the *atom:eq* call gives (do19)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{atom{:}eq}(\mathit{car}(y),\mathrm{T}))\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Pushing the seq back inside the *atom:eq* call gives (PST)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{car}(y)),\mathrm{T})\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Again eliminating the call to *car* produces (PST, seq2, do2, do1)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathit{atom{:}eq}(\mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathrm{NIL}),\mathrm{T})\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Pulling the seq out of the *atom:eq* for the last time leaves (PST)

$$\begin{aligned}
\simeq\ &\mathtt{let}\{\mathit{ans} \leftarrow \mathtt{seq}(\mathit{rplaca}(x,\mathrm{T}),\mathit{rplaca}(y,\mathrm{NIL}),\mathit{atom{:}eq}(\mathrm{NIL},\mathrm{T})\}\\
&\quad \mathtt{seq}(\mathit{rplaca}(x,\mathrm{x_a}),\mathit{rplaca}(y,\mathrm{y_a}),\mathit{ans}).
\end{aligned}$$

Simplifying the *atom:eq* call, (do21)

$\simeq$ let{*ans* ← seq(*rplaca*(*x*, T), *rplaca*(*y*, NIL), NIL)}
seq(*rplaca*(*x*, x_a), *rplaca*(*y*, y_a), *ans*).

Now pulling the seq out of the let, again for the last time, (PST)

$\simeq$ seq(*rplaca*(*x*, T),
rplaca(*y*, NIL),
let{*ans* ← NIL}seq(*rplaca*(*x*, x_a), *rplaca*(*y*, y_a), *ans*)).

We can now eliminate the let in favor of the following (let3, seq2)

$\simeq$ seq(*rplaca*(*x*, T), *rplaca*(*y*, NIL), *rplaca*(*x*, x_a), *rplaca*(*y*, y_a), NIL).

Commuting the calls to *rplaca* results in (do10, seq2)

$\simeq$ seq(*rplaca*(*x*, T), *rplaca*(*x*, x_a), *rplaca*(*y*, NIL), *rplaca*(*y*, y_a), NIL).

Cancelling the redundant modification of *y* yields (do5, do8, seq2)

$\simeq$ seq(*rplaca*(*x*, T), *rplaca*(*x*, x_a), NIL).

Similarly cancelling the modification of *x* provides (do5, do8, seq2)

$\simeq$ seq(NIL).

The simplest property of seq gives the desired result, (seq1)

$\simeq$ NIL.

□1. □**Theorem**

6.2. Example 2: An Iterative Append Program

Consider the following two versions of the list append *function*, the first being the traditional pure version.

append(u, v) ← if(u, *cons*(*car*(u), *append*(*cdr*(u), v)), v)

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

it:append(u, v) ← ifn(u, v, let{w ← *cons*(*car*(u), v)}seq(*it:app*(*cdr*(u), w, v), w)))
it:app(u, w, v) ←
ifn(u, u, let{z ← *cons*(*car*(u), v)}seq(*rplacd*(w, z), *it:app*(*cdr*(u), z, v)))

Notice that while this system of definitions is not tail recursive, it can easily be made so by adding an extra parameter val to *it:app*, modifying the definitions appropriately. The following result verifies the correctness of the *it:append* program.

Theorem: $it{:}append(\mathrm{u},\mathrm{v}) \simeq append(\mathrm{u},\mathrm{v})$

Definition: We define the set of lists, M_{list}, as follows.

$$v\,;\mu \in \mathsf{M}_{list} \leftrightarrow (\exists n \in \mathsf{N})(\partial_{v;\mu}(\underbrace{[1,\ldots,1]}_{n\ 1s}) = \mathtt{NIL}).$$

Thus $c\,;\mu$ is in M_{list} iff some *cdr*-chain leads to an atom and this atom is NIL.

Proof: Clearly if $\mathrm{u} = \mathtt{NIL}$ then the theorem is true, so suppose that $\mathrm{u} \in \mathsf{M}_{list} - \{\mathtt{NIL}\}$. We prove the following lemma by induction on the length of u.

Lemma: $append(\mathrm{u},\mathrm{v}) \simeq \mathtt{let}\{\mathtt{w} \leftarrow cons(car(\mathrm{u}),\mathrm{v})\}\mathtt{seq}(it{:}app(cdr(\mathrm{u}),\mathtt{w},\mathrm{v}),\mathtt{w})$

Proof of Lemma:

Base Case: $car(\mathrm{u}) = x$ and $cdr(\mathrm{u}) = \mathtt{NIL}$ here we have

$$\begin{aligned}
&\mathtt{let}\{\mathtt{w} \leftarrow cons(car(\mathrm{u}),\mathrm{v})\}\mathtt{seq}(it{:}app(cdr(\mathrm{u}),\mathtt{w},\mathrm{v}),\mathtt{w})\\
&\simeq \mathtt{let}\{\mathtt{w} \leftarrow cons(x,\mathrm{v})\}\mathtt{seq}(it{:}app(\mathtt{NIL},\mathtt{w},\mathrm{v}),\mathtt{w}),\\
&\simeq \mathtt{let}\{\mathtt{w} \leftarrow cons(x,\mathrm{v})\}\mathtt{seq}(\mathtt{NIL},\mathtt{w}),\\
&\simeq \mathtt{let}\{\mathtt{w} \leftarrow cons(x,\mathrm{v})\}\mathtt{w},\\
&\simeq cons(x,\mathrm{v})\\
&\simeq append(\mathrm{u},\mathrm{v})
\end{aligned}$$

□Base Case

Induction Step: Suppose $car(\mathrm{u}) = x$ and $cdr(\mathrm{u}) = \mathrm{u}_0$ with $\mathrm{u}_0 \in \mathsf{M}_{list} - \{\mathtt{NIL}\}$, then

$$\begin{aligned}
&\mathtt{let}\{\mathtt{w} \leftarrow cons(car(\mathrm{u}),\mathrm{v})\}\mathtt{seq}(it{:}app(cdr(\mathrm{u}),\mathtt{w},\mathrm{v}),\mathtt{w})\\
&\simeq \mathtt{let}\{\mathtt{w} \leftarrow cons(x,\mathrm{v})\}\mathtt{seq}(it{:}app(\mathrm{u}_0,\mathtt{w},\mathrm{v}),\mathtt{w}),
\end{aligned}$$

Now

$$\begin{aligned}
&\mathtt{seq}(it{:}app(\mathrm{u}_0,\mathtt{w},\mathrm{v}),\mathtt{w})\\
&\simeq \mathtt{seq}(\mathtt{let}\{\mathtt{z} \leftarrow cons(car(\mathrm{u}_0),\mathrm{v})\}\mathtt{seq}(rplacd(\mathtt{w},\mathtt{z}), it{:}app(cdr(\mathrm{u}_0),\mathtt{z},\mathrm{v})),\mathtt{w}),
\end{aligned}$$

by unfolding the *it:app* call.

$$\simeq \mathtt{seq}(\mathtt{let}\{\mathtt{z} \leftarrow cons(car(\mathrm{u}_0),\mathrm{v})\}\mathtt{seq}(it{:}app(cdr(\mathrm{u}_0),\mathtt{z},\mathrm{v}), rplacd(\mathtt{w},\mathtt{z})),\mathtt{w}),$$

since w is not visible in $it{:}app(cdr(\mathrm{u}_0),\mathtt{z},\mathrm{v})$.

Also note that

$$\mathtt{let}\{\mathtt{z} \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}_0), \mathrm{v})\}\mathtt{seq}(\mathit{it}{:}\mathit{app}(\mathit{cdr}(\mathrm{u}_0), \mathtt{z}, \mathrm{v})\ \mathit{rplacd}(\mathtt{w}, \mathtt{z}))$$

$$\simeq \mathtt{let}\{\mathtt{z} \leftarrow \mathtt{let}\{\mathtt{w}_0 \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}_0), \mathrm{v})\}\mathtt{seq}(\mathit{it}{:}\mathit{app}(\mathit{cdr}(\mathrm{u}_0), \mathtt{w}_0, \mathrm{v}), \mathtt{w}_0)\}\ \mathit{rplacd}(\mathtt{w}, \mathtt{z}),$$

by using some simple properties of the `let` construct.

$$\simeq \mathtt{let}\{\mathtt{z} \leftarrow \mathit{append}(\mathrm{u}_0, \mathrm{v})\}\ \mathit{rplacd}(\mathtt{w}, \mathtt{z}),$$

by the induction hypothesis.

Thus we have

$$\mathtt{let}\{\mathtt{w} \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}), \mathrm{v})\}\mathtt{seq}(\mathit{it}{:}\mathit{app}(\mathit{cdr}(\mathrm{u}), \mathtt{w}, \mathrm{v}), \mathtt{w})$$

$$\simeq \mathtt{let}\{\mathtt{w} \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}), \mathrm{v})\}\mathtt{seq}(\mathtt{let}\{\mathtt{z} \leftarrow \mathit{append}(\mathrm{u}_0, \mathrm{v})\}\ \mathit{rplacd}(\mathtt{w}, \mathtt{z}), \mathtt{w})$$

$$\simeq \mathtt{let}\{\mathtt{w} \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}), \mathrm{v})\}\mathtt{seq}(\mathit{rplacd}(\mathtt{w}, \mathit{append}(\mathrm{u}_0, \mathrm{v})), \mathtt{w}),$$

eliminating the inner `let` for something simpler.

$$\simeq \mathtt{let}\{\mathtt{w} \leftarrow \mathit{cons}(\mathit{car}(\mathrm{u}), \mathrm{v})\}\ \mathit{rplacd}(\mathtt{w}, \mathit{append}(\mathrm{u}_0, \mathrm{v})),$$

$$\simeq \mathit{rplacd}(\mathit{cons}(\mathit{car}(\mathrm{u}), \mathrm{v}), \mathit{append}(\mathrm{u}_0, \mathrm{v})),$$

$$\simeq \mathit{cons}(\mathit{car}(\mathrm{u}), \mathit{append}(\mathrm{u}_0, \mathrm{v})),$$

$$\simeq \mathit{append}(\mathrm{u}, \mathrm{v})$$

□**Lemma**

The theorem now follows by observing that if $u, v\,;\mu \in \mathsf{M}_{sexp}$ with $u\,;\mu \notin \mathsf{M}_{list}$ then neither $\mathit{append}(u, v)\,;\mu$ nor $\mathit{it}{:}\mathit{append}(u, v)\,;\mu$ will denote. □**Theorem**

6.3. Example 3: The Deutsch-Schorr-Waite Marking Algorithm

We chose this example for historical reasons, since there have been many different verification proofs in the literature. It is a version of the famous Deutsch-Schorr-Waite marking algorithm, (Schorr and Waite, 1967) and (Deutsch, 1968), that is used in implementing mark and sweep garbage collection, one method for reclaiming useless and unusable *cons* cells in the usual dynamic storage allocation environment. See (Knuth, 1968) or (Schorr and Waite, 1967) for an extensive discussion of garbage collection and marking algorithms. In this example we describe what we believe to be the most convincing, useful and least technical proof. We shall also discuss the other proofs since they all differ from this one. In this example we work within the low level Lisp data structure M_{mfsexp} introduced earlier. The Deutsch-Shorr-Waite marking algorithm uses pointer reversal to avoid using an explicit stack. Pointer reversal is a very powerful technique that is used in destructive memory programming. The idea is quite simple: the program destructively alters the structure it is operating on to store the information that a stack would normally be used for. In this case the algorithm scans the graph in a left-first fashion, marking cells as it proceeds. Since the cells are marked when they are first visited,

looping or repeatedly scanning the same subgraph is avoided. A succinct treatment of pointer reversal, or pointer rotation as it is sometimes called, may be found in (Suzuki, 1982). In this structure our version of the algorithm is given by the following definition.

```
mark(c) ← mark0(c,NIL))
mark0(c,s) ←
    if(tm(c), pop(c,s), let{a ← car(c)}seq(setm(c,1), rplaca(c,s), mark0(a,c)))
pop(c,s) ←
    ifn(s,
        c,
        if(eq(f(s),0),
            let{d ← cdr(s), os ← car(s)}
                seq(setf(s,1), rplaca(s,c), rplacd(s,os), mark0(d,s))
            let{os ← cdr(s)}seq(setf(s,0), rplacd(s,c), pop(s,os))))
tm(c) ← or(atom(c), eq(m(c),1))
```

Our verification proof consists in showing that *mark* is strongly isomorphic to a very simple recursive program. This has several advantages over most proofs. Firstly, rather than proving that the algorithm meets an equally complicated theoretical specification, our specification, being the simple recursive version below, is by its very nature simpler and does not force the casual reader to grapple with the sometimes overbearing details of the underlying semantics and proof rules. Another reason is that our proof emphasizes the relationship between program verification, program transformation and program derivation, in that because the proof is of a syntactical nature it can be thought of in two ways. The first is the way we are presenting it, a verification of a complex algorithm via a reduction to a very simple algorithm. The second is the reverse or dual, obtaining or deriving a complex more efficient algorithm from a simpleminded or *specifying* algorithm. This very simple algorithm is defined as follows:

```
mark1(c) ← if(tm(c), c, seq(setm(c,1), mark1(car(c)), mark1(cdr(c)), c))
```

As we have already stated, our verification proof simply involves showing that the following holds under certain simple conditions:

Theorem: $mark(x) \simeq mark1(x)$.

Of the existing proofs in the literature the closest to ours in spirit, but not in style, is that found in (Topor, 1979). Topor's approach is to separate the proof into two essentially distinct parts. The first part involves properties of the underlying data structure, while the second relies on properties of the algorithm itself. The proof itself is of the *intermittent assertions* variety that was first introduced by (Knuth, 1968) and later developed by (Burstall, 1974) and (Manna and Waldinger, 1977). Topor's paper is also interesting in that there he, to some extent, suggests the possibility of our proof, or at least one like it. In (Topor, 1979) he gives three marking algorithms, Algorithm 1, which is a slight variation on our *mark1*, Algorithm 3, being his version of the Deutsch-Schorr-Waite

marking algorithm, and in between, Algorithm 2, which we describe by the following definition:

$$mark2(\mathtt{c}) \leftarrow \mathtt{seq}(m2(\mathtt{c}, \mathtt{NIL}), \mathtt{c})$$

$$m2(\mathtt{c}, \mathtt{s}) \leftarrow$$
$$\mathtt{if}(tm(\mathtt{c}), \mathtt{if}(\mathtt{s}, m2(cdar(\mathtt{s}), cdr(\mathtt{s})), \mathtt{c}), \mathtt{seq}(setm(\mathtt{c}, 1), m2(car(\mathtt{c}), cons(\mathtt{c}, \mathtt{s}))))$$

This version is simply a(n almost) tail recursive program obtained from the trivial one by the incorporation of an explicit stack. *mark* eliminates this storage extravagant feature by storing the stack on the structure being marked. In his conclusions Topor says:

> *An alternative method of proof which has apparently not been used is to first verify Algorithm 2, a purely constructive program, using the present (or any other) method, and then use the techniques of (Milner, 1971) or (Hoare, 1972) to show that the Shorr-Waite algorithm simulates Algorithm 2. In such a proof the representation function describing how the stack of Algorithm 2 is represented in the Schorr-Waite algorithm is defined by*

$$\begin{aligned}
&\mathtt{Rep}(\mathtt{X}) \leftarrow \\
&\quad \mathbf{if}\ \mathtt{X} = \mathtt{NIL}\ \mathbf{then}\ \mathtt{empty} \\
&\quad \mathbf{else\ if}\ \mathtt{f}(\mathtt{X}) = 0\ \mathbf{then}\ \mathtt{Rep}(\mathtt{car}(\mathtt{X})) \\
&\quad \mathbf{else}\ \mathtt{Push}(\mathtt{X}, \mathtt{Rep}(\mathtt{car}(\mathtt{X}))).
\end{aligned}$$

> *However the structure of this definition means that the proof of simulation is not completely straightforward but requires yet another inductive argument. Since the proof of Algorithm 2 is almost as long as that of the Schorr-Waite algorithm, the resulting proof is no shorter than the one we have given. This fact suggests that further research into proofs of simulation and the use of data structure is desirable.*

This suggestion is very much in the spirit of our proof. The main differences are that we do not choose Algorithm 2 as our *go-between* but rather the trivial Algorithm 1. Thus our proof is not one of *simulation* but of *actual equivalence*, and does not suffer from the deficiencies that Topor raises. We also do not use the methods of (Milner, 1971) or (Hoare, 1972) but our own.

Most of the other proofs of correctness are of the *inductive assertions* variety as described by (Floyd, 1967). Gries, in a companion paper to Topor's (Gries, 1979), proves the correctness and termination of a version of the algorithm in a rather elegant fashion, although he simplifies the world so as to combine the *mark* and *field* bits as well as exclude the existence of atoms. Other proofs have been given. We mention them briefly: Kowaltowski (Kowaltoski, 1973) extended the techniques of (Burstall, 1972) to encompass a proof of the marking algorithm; Suzuki (Suzuki, 1976) in his dissertation, describes a machine checked proof that uses recursively defined predicates in his assertions; Morris (Morris, J. H, 1972) gives a more direct analysis similar to that of Topor's; Dershowitz (Dershowitz, 1980) derives, informally, the algorithm and its correctness from a simple algorithm via an intermediate version; more formal derivations of these and other, more

complex algorithms are given in (Lee, Gerhart and deRoever, 1979) and (van Diepen and deRoever, 1986); finally a proof using denotational semantics has been given in (Poupon and Wegbreit, 1972). Let $\mathbf{U}(\mathtt{c})$ be the set of unmarked cells reachable from $\mathtt{c}$ via paths through unmarked cells. We finish this example by proving

Theorem: $mark(x) \simeq mark1(x)$, assuming $(\forall c \in \mathbf{U}(x)) f(c) = 0$.

Lemma: $mark0(\mathtt{c},\mathtt{s}) \simeq \mathtt{seq}(mark1(\mathtt{c}), pop(\mathtt{c},\mathtt{s}))$ under the hypothesis of the theorem.

To see that the theorem follows it suffices to observe that $mark0(\mathtt{c},\mathtt{NIL}) \simeq mark1(\mathtt{c})$ and that $mark1(\mathtt{c}) \simeq \mathtt{seq}(mark1(\mathtt{c}), pop(\mathtt{c},\mathtt{NIL}))$. The proof of the lemma is straightforward, using the following simple commutativity property of *mark1*. This is left to the reader to verify. It expresses the fact that *mark1* does not depend on the contents of a marked cell.

Commutativity Property: If $\vartheta \in \{rplaca, rplacd, setf\}$, $e \in \mathbf{E}$ and $m(x) \simeq 1$, then

$$\mathtt{seq}(mark1(z), \vartheta(x,y), e) \simeq \mathtt{seq}(\vartheta(x,y), mark1(z), e).$$

Proof of lemma: We prove this lemma by induction on the size of $\mathbf{U}(\mathtt{c})$. For convenience we suppose that $m(\mathtt{c}) = 0$, $car(\mathtt{c}) = \mathtt{c_a}$ and $cdr(\mathtt{c}) = \mathtt{c_d}$. Since the identity clearly holds when $\mathtt{c}$ is terminal, we also assume that $f(c) = 0$ whenever $c \in \mathbf{U}(\mathtt{c})$.

$$mark0(\mathtt{c},\mathtt{s})$$

$$\simeq \mathtt{if}(tm(\mathtt{c}), pop(\mathtt{c},\mathtt{s}), \mathtt{let}\{\mathtt{a} \leftarrow car(\mathtt{c})\}\mathtt{seq}(setm(\mathtt{c},1), rplaca(\mathtt{c},\mathtt{s}), mark0(\mathtt{a},\mathtt{c}))).$$

Evaluating the `let` gives

$$\simeq \mathtt{if}(tm(\mathtt{c}), pop(\mathtt{c},\mathtt{s}), \mathtt{seq}(setm(\mathtt{c},1), rplaca(\mathtt{c},\mathtt{s}), mark0(\mathtt{c_a},\mathtt{c}))).$$

Since $\mathtt{c}$ has been marked we must have $|\mathbf{U}(\mathtt{c_a})| < |\mathbf{U}(\mathtt{c})|$ thus by induction:

$$\simeq \mathtt{if}(tm(\mathtt{c}), pop(\mathtt{c},\mathtt{s}), \mathtt{seq}(setm(\mathtt{c},1), rplaca(\mathtt{c},\mathtt{s}), mark1(\mathtt{c_a}), pop(\mathtt{c_a},\mathtt{c}))).$$

Unfolding the $pop(\mathtt{c_a},\mathtt{c})$ call and eliminating the vacuous `ifn` yields

$$\begin{aligned}
\simeq\ & \mathtt{if}(tm(\mathtt{c}),\\
& \quad pop(\mathtt{c},\mathtt{s}),\\
& \quad \mathtt{seq}(setm(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_a})\\
& \qquad \mathtt{if}(eq(f(\mathtt{c}),0),\\
& \qquad\quad \mathtt{let}\{\mathtt{d} \leftarrow cdr(\mathtt{c}), \mathtt{os} \leftarrow car(\mathtt{c})\}\\
& \qquad\qquad \mathtt{seq}(setf(\mathtt{c},1), rplaca(\mathtt{c},\mathtt{c_a}), rplacd(\mathtt{c},\mathtt{os}), mark0(\mathtt{d},\mathtt{c}))\\
& \qquad\quad \mathtt{let}\{\mathtt{os} \leftarrow cdr(\mathtt{c})\}\mathtt{seq}(setf(\mathtt{c},0), rplacd(\mathtt{c},\mathtt{c_a}), pop(\mathtt{c},\mathtt{os})))).
\end{aligned}$$

This simplifies to the following since the field bit is 0

$$
\begin{aligned}
\simeq\ & \mathtt{if}(tm(\mathtt{c}),\\
& \quad pop(\mathtt{c},\mathtt{s}),\\
& \quad \mathtt{seq}(setm(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_a}),\\
& \qquad setf(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{c_a}),\\
& \qquad rplacd(\mathtt{c},\mathtt{s}),\\
& \qquad mark0(\mathtt{c_d},\mathtt{c}))).
\end{aligned}
$$

Again we can use the induction hypothesis to obtain

$$
\begin{aligned}
\simeq\ & \mathtt{if}(tm(\mathtt{c}),\\
& \quad pop(\mathtt{c},\mathtt{s}),\\
& \quad \mathtt{seq}(setm(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_a}),\\
& \qquad setf(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{c_a}),\\
& \qquad rplacd(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_d}),\\
& \qquad pop(\mathtt{c_d},\mathtt{c}))).
\end{aligned}
$$

Unfolding and simplifying the $pop(\mathtt{c_d},\mathtt{c})$ call results in

$$
\begin{aligned}
\simeq\ & \mathtt{if}(tm(\mathtt{c}),\\
& \quad pop(\mathtt{c},\mathtt{s}),\\
& \quad \mathtt{seq}(setm(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_a}),\\
& \qquad setf(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{c_a}),\\
& \qquad rplacd(\mathtt{c},\mathtt{s}),\\
& \qquad mark1(\mathtt{c_d}),\\
& \qquad setf(\mathtt{c},0),\\
& \qquad rplacd(\mathtt{c},\mathtt{c_d}),\\
& \qquad pop(\mathtt{c},\mathtt{s}))).
\end{aligned}
$$

Using the commutativity property gives

$$
\begin{aligned}
\simeq\ & \mathtt{if}(tm(\mathtt{c}),\\
& \quad pop(\mathtt{c},\mathtt{s}),\\
& \quad \mathtt{seq}(setm(\mathtt{c},1),\\
& \qquad rplaca(\mathtt{c},\mathtt{s}),\\
& \qquad rplaca(\mathtt{c},\mathtt{c_a}),\\
& \qquad setf(\mathtt{c},1),\\
& \qquad setf(\mathtt{c},0),\\
& \qquad rplacd(\mathtt{c},\mathtt{s}),\\
& \qquad rplacd(\mathtt{c},\mathtt{c_d}),\\
& \qquad mark1(\mathtt{c_a}),\\
& \qquad mark1(\mathtt{c_d}),\\
& \qquad pop(\mathtt{c},\mathtt{s}))).
\end{aligned}
$$

Finally cancelling redundant operations gives

$$\simeq \mathtt{if}(tm(\mathtt{c}), pop(\mathtt{c},\mathtt{s}), \mathtt{seq}(setm(\mathtt{c},1), mark1(\mathtt{c_a}), mark1(\mathtt{c_d}), pop(\mathtt{c},\mathtt{s}))).$$

We obtain the result by folding and a minimum of effort

$$\simeq \mathtt{seq}(mark1(\mathtt{c}), pop(\mathtt{c},\mathtt{s})).\square_{\mathbf{Lemma}}\square_{\mathbf{Theorem}}$$

7. Conclusions

This paper has provided a precise but elegant framework for reasoning about programs which destructively manipulate their data. In the foundational respect it is entirely new. It is an implicit aim of this paper to show that the verification, specification and transformation of programs are not, and should not be, distinct enterprises. Thus we view our work as a contribution, however small, to *inferential programming.*

The theory the paper describes is relatively complete insofar as it achieves its initial aims. It is, however, a mere stepping stone in the whole scheme of things. In (Mason, 1986) we adapted this framework to a transformation system, based on that of (Scherlis, 1980). Within this framework one can easily derive, for example, the *it:append* program from the simple pure one. More work needs to be done, however, before such a system can be automated and, perhaps, used by a optimizing compiler. In (Manna and Waldinger, 1987) the framework is adapted to the deductive synthesis of programs from specifications. Yet again this work is in an early stage of development. Nevertheless these practical applications seem promising.

We finish by pointing out further directionsof research, other than those above. The area of research that we are most immediately interested in to follow up on this paper is that of incorporating high level constructs into our language, and enlarging the framework accordingly. Two examples are:

1. Closures, allowing functions or functionals to be passed as arguments and returned as values.

2. Continuations, allowing for non-functional control constructs that make continuations, i.e. the remaining part of the computation, first class objects. There is good reason to believe that this will be somewhat more difficult than the previous problem.

It is our belief that the most fruitful approach that can be taken in these examples is the study of the *equivalence* of programs, to prove that programs with these high level constructs are equivalent to programs without them. The simple programs play the role of specifications and the proofs of equivalence the role either as verifications or derivations of the more complex programs. Operations or transformations on programs from one language to another are central to the whole approach. Operations on programs need meanings to transform and meanings to preserve and the study of various notions of equivalence is simply a study of the various *meanings*. This variety of languages and interpretations should exist within a single framework. They should be *compatible* in the sense that the value returned by a program should be the same in each interpretation. They should also be *coherent* in that one can move gracefully between interpretations and have systematic methods for *deriving* one interpretation from another. The fact the target languages and specification languages are supersets or subsets of one another permits special interpretation of fragments. This approach emphasizes the role of transformation, the new paradigm; in the long term it is a contribution to inferential programming, seen as the unification of programming, program verification, program derivation and program transformation.

Acknowledgements I thank C.L.Talcott and the referees for many useful remarks on an earlier version of this work.

8. Bibliography

Bloom, S. and Tindell, R. 1983, *Varieties of if-then-else*. SIAM Journal of Computing, Volume 12. pp 677-707.

Burstall, R. M. 1972, *Some Techniques for Proving Correctness of Programs which Alter Data Structures*. in Meltzer, B. and Mitchie, D. (eds.) *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, Scotland. pp. 23-50.

Burstall, R. M. 1974, *Program Proving as Hand Simulation with a Little Induction*. Proceedings IFIP Congress, North Holland. pp 308-312.

Burstall, R. M. and Darlington, J. 1977, *A Transformation System for Developing Recursive Programs*. Journal of the ACM, vol 24, no. 1, January.

Dershowitz, N. 1980, *The Schorr–Waite Marking Algorithm Revisited*. Information Processing Letters, vol 11, no. 3, November.

Deutsch, L. P. 1969, p 417, Volume 1, (Knuth, 1968).

van Diepen, N.W.P. and deRoever, W.P. 1986, *Program Derivation through Transformations: The Evolution of List–Copying Algorithms*. Science of Computer Programming , vol 6, pp 213-272.

Floyd, R. W. 1967, *Assigning Meanings to Programs.* Proceedings of the Symposium in Applied Mathematics 19. (J.T.Schwartz, ed.) A.M.S, pp 19-32.

Goad, C. A. 1980, *Computational Uses of the Manipulation of Formal Proofs.* Stanford Computer Science Report No. STAN-CS-80-819.

Guessarian, I. and Meseguer, J. 1985, *On the Axiomatization of if-then-else.* CSLI Report No. CSLI-85-20.

Gries, D. 1979, *The Schorr-Waite Graph Marking Algorithm.* Acta Informatica, Volume 11, pp 223-232.

Hoare, C. A. R. 1972, *Proof of Correctness of Data Representations.* Acta Informatica, Volume 1, pp 271-281.

Kowaltowski, T. 1973, *Correctness of Programs Manipulating Data Structures.* PhD thesis, University of California, Berkeley.

Knuth, D. E. 1968, *The Art of Computer Programming.* Volume 1. Addison-Wesley, London.

Lee, S., Gerhart, S.L. and deRoever, W.P. 1979, *The Evolution of List–Copying Algorithms.* Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages.

Manna, Z. and Waldinger, R. 1977, *Is Sometimes Better Than Always? Intermittent Assertions in Proving Program Correctness.* Communications of the ACM, Volume 21, pp 159-172.

Manna, Z. and Waldinger, R. 1987, *The Deductive Synthesis of Imerative LISP Programs.* Proceedings of the AAAI.

Mason, I. A. 1986, *The Semantics of Destructive Lisp.* CSLI Lecture Notes, Number 5, The University of Chicago Press, Chicago.

Mason, I. A. and Talcott, C. L. 1985, *Memories of S-expressions.* Stanford Computer Science Report No. STAN-CS-85-1057.

Milner, R. 1971, *An Algebraic Definition of Simulation Between Programs.* Stanford Computer Science Report No. STAN-CS-71-205.

Morris, J. H. Jr. 1972, *Verification-Orientated Language Design.* Technical report 7. Department of Computer Science, University of California, Berkeley.

Oppen, D. C. 1978, *Reasoning about Recursively Defined Data Structures.* Stanford AIM-314.

Poupon, J. and Wegbreit, B. 1972, *Covering functions.* Unpublished report, Center for research in computing technology, Harvard university.

Scherlis, W. L. 1980, *Expression Procedures and Program Derivation.* Stanford Com-

puter Science Report No. STAN-CS-80-818.

Scherlis, W. L. and Scott, D. S. 1983, *First Steps Towards Inferential Programming.* Information Processing 83, R.E.A. Mason (ed), North Holland, New York.

Schorr, H. and Waite, W. M. 1967, *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures.* Communications of the ACM, Volume 10, pp 501-506.

Suzuki, N. 1976, *Automatic Verification of Programs with Complex Data Structures.* Ph.D. thesis, Stanford University.

Suzuki, N. 1982, *Analysis of Pointer Rotation.* Communications of the ACM, Volume 25, Number 5, pp 330-335.

Talcott, C. L. 1985, *The Essence of RUM.* Stanford Computer Science Report No. STAN-CS-85-1060.

Talcott, C. L. 1986, *Program and Proving with Function and Control Abstractions.* Unpublished Manuscript.

Tenney. 1972, *Decidable Pairing Functions.* PhD Thesis, Cornell University.

Topor, R. W. 1979, *The Correctness of the Shorr-Waite List Marking Algorithm.* Acta Informatica, Volume 11, pp 211-221.

Copyright © 1987, Laboratory for Foundations of Computer Science,
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.