Memories of S-Expressions Proving Properties of Lisp-Like Programs That Destructively Alter Memory

ARTICLE · MAY 1985

CITATION
1

READS
10

2 AUTHORS, INCLUDING:

Ian A. Mason
SRI International
60 PUBLICATIONS **1,536** CITATIONS

SEE PROFILE

June 1985

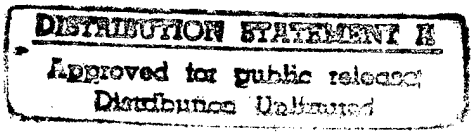
Report No. STAN-CS-85-1057

PB96-148861

Memories of S-expressions
Proving properties of Lisp-like programs that destructively alter memory

by

Ian A. Mason and Carolyn L. Talcott

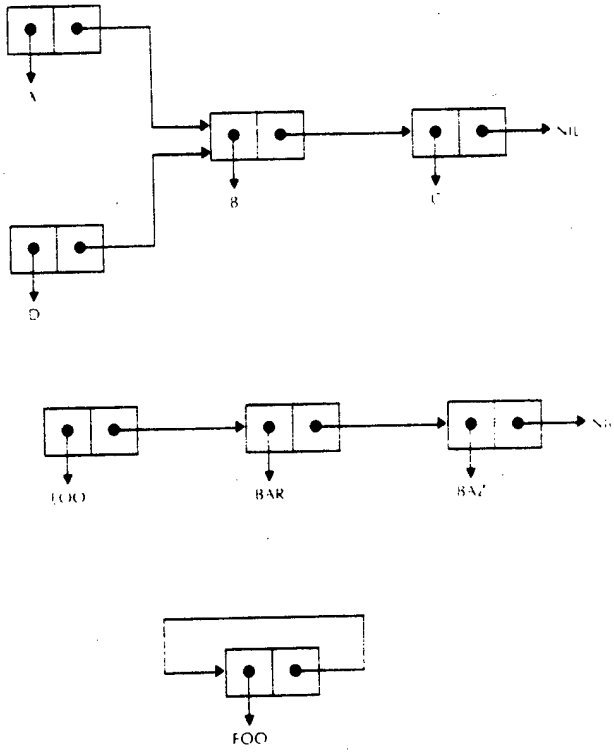


Department of Computer Science

Stanford University
Stanford, CA 94305

Memories of S-expressions

Proving properties of Lisp-like programs that destructively alter memory



Ian A. Mason and Carolyn L. Talcott

Stanford University

Research supported by ARPA contract N00039-82-C-0250.

1. Introduction and Notation

In this paper we present a mathematical model called a memory structure, and define a computation theory over such structures. This computation theory provides a semantics for first-order, lexically scoped Lisp-like languages and we use this as a basis for expressing and proving properties of a variety of programs that destructively alter the contents of memory. Since we have chosen to work in a Lisp-like world, our subject matter is particularly relevant to the Lisp programmer. The main example in this paper is a proof of the correctness of the Robson copying algorithm, [R]. This algorithm copies possibly cyclic Lisp style S-expressions using bounded storage and illustrates how destructive memory operations can be used to write fast efficient programs. The paper is organized as follows: In section two we describe the class of memory structures and introduce $\mathbb{M}_{sexp}$, the S-expression memory structure as a particular example. In section three we describe a computation theory over these structures that corresponds to a Lisp-like programming language. In section four we study $\mathbb{M}_{sexp}$ in more detail, developing the concepts one usually finds in Lisp-like languages. Section five gives four simple correctness proofs of typical Lisp programs both destructive and otherwise. The last two sections deal with the Robson copying algorithm and related programs.

We finish off this section by describing some of our notation. We use the usual notation for set membership and function application. Let $\mathbb{D}$, $\mathbb{D}_0$, $\mathbb{D}_1$, $\ldots\mathbb{D}_n$ be sets, then $\mathbb{D}_0 \oplus \mathbb{D}_1$ is the (disjoint) union of $\mathbb{D}_0$ and $\mathbb{D}_1$. We only use $\oplus$ applied to disjoint sets, thus it is mainly a matter of emphasis. $\mathbb{D}_0 \otimes \ldots \otimes \mathbb{D}_{n-1}$ is the set of n-tuples with i^{th} element from $\mathbb{D}_i$ for $i < n$. We write $\mathbb{D}^{(n)}$ for $\mathbb{D}_0 \otimes \ldots \otimes \mathbb{D}_{n-1}$ when each $\mathbb{D}_i$ is $\mathbb{D}$. $\mathbb{D}^*$ is the set of finite sequences of elements of $\mathbb{D}$.

$$\mathbb{D}^* = \bigcup_{n \in \omega} \mathbb{D}^{(n)}$$

Some notation for sequences follows. $\square$ is the empty sequence, the unique element of $\mathbb{D}^{(0)}$ for any domain $\mathbb{D}$. For $d, d_0, \ldots, d_{n-1}, d'_0, \ldots, d'_{m-1} \in \mathbb{D}$, the sequence of length n with i^{th} element d_i for $i < n$ is written $[d_0, \ldots, d_{n-1}]$. Let $v = [d_0, \ldots, d_{n-1}]$, $u = [d'_0, \ldots, d'_{m-1}]$ and $i < n$ then $|v|$ is the length of v while $v\downarrow_i$ is the i^{th} element of v, namely d_i. $v \diamond u \underset{\text{df}}{=} [d_0, \ldots, d_{n-1}, d'_0, \ldots, d'_{m-1}]$ is the concatenation of v and u. We identify d with the singleton sequence $[d]$. Note that $(u \diamond v) \diamond w = u \diamond (v \diamond w)$ and $[\ ] = \square$.

$\mathbf{P}_\omega\, \mathbb{D}$ is the domain of finite sets from $\mathbb{D}$. $[\mathbb{D}_0 \to \mathbb{D}_1]$ is the set of total functions from $\mathbb{D}_0$ to $\mathbb{D}_1$, and $[\mathbb{D}_0 \rightharpoonup \mathbb{D}_1]$ is the set of partial functions. If $\mu \in [\mathbb{D}_0 \rightharpoonup \mathbb{D}_1]$, then δ_μ is the domain of μ and ρ_μ is its range. For $d_0 \in \mathbb{D}_0, d_1 \in \mathbb{D}_1$, and $\mu \in [\mathbb{D}_0 \rightharpoonup \mathbb{D}_1]$ we let

$$\mu\{d_0 \leftarrow d_1\}$$

be the map μ_0 such that $\mu_0(d_0) = d_1$ and $\mu_0(d) = \mu(d)$ for $d \neq d_0$.

Some particular sets that we shall use frequently are as follows. $\mathbb{Z}$ is the integers and $z, z_0, \ldots$ range over $\mathbb{Z}$. $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the natural numbers and $n, n_0, \ldots$ range over $\mathbb{N}$. We consider a natural number to be the set of numbers less than it, thus the less than relation, $<$, is simply the membership relation, $\in$, of set theory. We let $\mathbb{T} = \{0, 1\}^*$ be the

complete binary tree, i.e. the set of finite sequences of 0's and 1's. We use 1^n to denote the sequence in $\mathbb{T}$ that consists of exactly n ones. Note that $1^0 = \square$. We shall adopt the convention that trees grow downward and $\sigma, \sigma_0, \ldots$ will range over $\mathbb{T}$. We use two partial orderings on $\mathbb{T}$. The initial segment relation, $<$, and the Brouwer-Kleene linear ordering, $\prec$. $\sigma_0 < \sigma_1$ is taken to mean that σ_1 is *below* σ_0 in $\mathbb{T}$, while $\sigma_0 \prec \sigma_1$ means that σ_0 is *before* σ_1 in $\mathbb{T}$. The *below* relation is defined by

$$\sigma_0 < \sigma_1 \leftrightarrow \exists \sigma \neq \square\, (\sigma_1 = \sigma_0 \diamond \sigma)$$

and the *before* relation is defined by

$$\sigma_0 \prec \sigma_1 \leftrightarrow \sigma_0 < \sigma_1 \vee \exists \sigma, \sigma_2, \sigma_3 (\sigma_0 = \sigma \diamond 0 \diamond \sigma_2 \wedge \sigma_1 = \sigma \diamond 1 \diamond \sigma_3).$$

The *before* relation is also known as the depth-first ordering.

We wish to thank several people, Ross Casley and Martin Ross for proofreading an earlier draft of this paper and detecting many absurdities, Dave Touretzky for kindly allowing us to reproduce some diagrams from his book [To], and finally Dennis de Champeaux for providing us with an annotated Interlisp version of the Robson copy algorithm, the reason this paper exists.

2. Memory Structures and $\mathbb{M}_{sexp}$

In this section we introduce the notion of a memory structure over a set $\mathbb{A}$ 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 (at any particular time this collection will of course be finite). The machine uses these locations to store various types and quantities of objects. There are machine instructions for accessing and updating the contents of memory locations. 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 will assume that our hypothetical machine can store a sequence of objects, (the sequence being of arbitrary finite length) each object being either an atom from $\mathbb{A}$ 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 will abstract away from these implementation dependent details.

In this paper we will be mainly concerned with S-expression memories that can only store pairs of objects in each location, however we will treat the general case first leaving S-expression memory structures as a particular example. Let $\mathbb{A}$ be some fixed set of atoms and $\mathbb{L}$ some countably infinite set disjoint from $\mathbb{A}$. $\mathbb{L}$ is the set of memory locations of our hypothetical machine. The elements of the sequences that are stored in these locations are

the *memory values* and we denote them by $\mathbb{V}$. Thus $\mathbb{V} = \mathbb{A} \oplus \mathbb{L}$. A memory μ is a function from a finite subset of $\mathbb{L}$ to the set of sequences of memory values, $\mathbb{V}^* = (\mathbb{A} \oplus \mathbb{L})^*$. Since we wish $\mu(l)$ to represent the contents of the location l in the memory μ, we also require that those locations which occur amongst the contents of locations are also locations in our memory. Thus we define a *memory* μ to be a finite map such that

$$\mu \in [\delta_\mu \twoheadrightarrow (\delta_\mu \oplus \mathbb{A})^*].$$

where δ_μ is a finite subset of $\mathbb{L}$. The set of all memories over $\mathbb{A}$ and $\mathbb{L}$ is denoted by $\mathbb{M}_{(\mathbb{A},\mathbb{L})}$.

Now suppose that $\mathbb{M}$ is a set of memories, a *memory object* of $\mathbb{M}$ is a pair

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

such that μ is a memory in $\mathbb{M}$ and the sequence $[v_0, \ldots, v_{n-1}]$ satisfies $v_i \in \delta_\mu \oplus \mathbb{A}$ for $i \in n$. Thus a memory object is a memory together with a sequence of memory values which *exist* in that memory. We invariably write such a memory object simply as $v_0, \ldots, v_{n-1} \,;\, \mu$. A *memory structure* is defined to be a set of memories $\mathbb{M}$ together with a set of operations $\mathbb{O}$, which are allowed to be partial, on those memory objects of $\mathbb{M}$. The operations model the machine instructions for manipulating objects. We usually refer to a memory structure by its collection of memories $\mathbb{M}$, taking the operations to be implicit. We also abuse notation and refer to the set of memory objects of a particular collection of memories $\mathbb{M}$ simply by $\mathbb{M}$, context should always prevent confusion. One last abuse of notation is that by $\mathbb{M}^{(n)}$ we *always* mean the collection of memory objects whose sequence of memory values is of length n, the reason for this is that 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. For ease of reading we let $\mu, \mu_0, \ldots$ range over memories, $v, v_0, \ldots$ range over $\mathbb{V}$, $a, a_0, \ldots$ range over $\mathbb{A}$ and $l, l_0, \ldots$ range over $\mathbb{L}$.

2.1. Definition of a memory structure $\mathbb{M}$

We can summarize the above definitions as follows:

- $\mathbb{A}$ and $\mathbb{L}$ are disjoint sets, $\mathbb{L}$ countable, and $\mathbb{V} = \mathbb{A} \oplus \mathbb{L}$ is the set of memory values.
- A memory is a finite map μ from $\mathbb{L}$ to $\mathbb{V}^*$ such that $\mu \in [\delta_\mu \twoheadrightarrow (\delta_\mu \oplus \mathbb{A})^*]$. The set of all memories over $\mathbb{A}$ and $\mathbb{L}$ is denoted by $\mathbb{M}_{(\mathbb{A},\mathbb{L})}$.
- Let $\mathbb{M}$ be a set of memories. A memory object of $\mathbb{M}$ is a tuple $v_0, \ldots, v_{n-1} \,;\, \mu$ such that μ is a memory in $\mathbb{M}$ and $v_i \in \delta_\mu \oplus \mathbb{A}$ for $i \in n$. We write $v_0, \ldots, v_{n-1} \,;\, \mu \in \mathbb{M}^{(n)}$ to emphasize the length of the memory value sequence.
- A memory structure is a set of memories $\mathbb{M}$ together with a set of operations $\mathbb{O}$ on memory objects of $\mathbb{M}$.

2.2. 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 assume that the integers $\mathbb{Z}$ are contained in $\mathbb{A}$ and that $\mathbb{A}$ contains two non-numeric atoms T and NIL. These atoms are used to represent *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 we 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, $\mathbb{M}_{sexp}$, is defined by:

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

Thus, as we mentioned earlier, the S-expression memory can only store pairs of memory values in its memory locations. To complete our specification of of the S-expression memory structure we need only describe the operations $\mathbb{O}_{sexp}$. These are as follows:

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

int? and *cons*? are characteristic functions (recognizers) of $\mathbb{Z}$ and $\mathbb{L}$, and *eq* is the characteristic function of equality.

$$int?(v\,;\mu) = \begin{cases} \mathtt{T}\,;\mu & \text{if } v \in \mathbb{Z} \\ \mathtt{NIL}\,;\mu & \text{if } v \notin \mathbb{Z} \end{cases}$$

$$cons?(v\,;\mu) = \begin{cases} \mathtt{T}\,;\mu & \text{if } v \in \mathbb{L} \\ \mathtt{NIL}\,;\mu & \text{if } v \notin \mathbb{L} \end{cases}$$

$$eq(v_0, v_1\,;\mu) = \begin{cases} \mathtt{T}\,;\mu & \text{if } v_0 = v_1 \\ \mathtt{NIL}\,;\mu & \text{if } v_0 \neq v_1 \end{cases}$$

add1 and *sub1* are the successor and predecessor functions on $\mathbb{Z}$.

$$add1(z\,;\mu) = z + 1\,;\mu$$

$$sub1(z\,;\mu) = z - 1\,;\mu$$

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 object by *selecting* a new location from *free storage* and putting the arguments as its contents. The free storage of a memory μ is just another name for $\mathbb{L} - \delta_\mu$.

$$cons(v_0, v_1\,;\mu) = l\,;\mu_0 \qquad \text{where} \quad l \notin \delta_\mu \quad \text{and} \quad \mu_0 = \mu\{l \leftarrow [v_0, v_1]\}$$

$$car(l\,;\mu) = \mu(l)\downarrow_0\,;\mu$$

$$cdr(l\,;\mu) = \mu(l)\downarrow_1\,;\mu$$

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.

$$\textit{rplaca}(l, v\,;\mu) = l\,;\mu_0 \qquad \text{where} \quad \mu_0 = \mu\{l \leftarrow [v, \mu(l)\downarrow_1]\}$$
$$\textit{rplacd}(l, v\,;\mu) = l\,;\mu_0 \qquad \text{where} \quad \mu_0 = \mu\{l \leftarrow [\mu(l)\downarrow_0, v]\}$$

We shall refer to the S-expression memory structure simply by $\mathbb{M}_{sexp}$.

In most cases we shall not be interested in the value of the *rplacx* operations, $x \in \{a, d\}$, so for convenience we define the operations *setcar* and *setcdr*.

$$\textit{setcar}(l, v\,;\mu) = \mu\{l \leftarrow [v, \mu(l)\downarrow_1]\}$$
$$\textit{setcdr}(l, v\,;\mu) = \mu\{l \leftarrow [\mu(l)\downarrow_0, v]\}$$

Note that $\textit{rplacx}(l, v\,;\mu) = l\,;\textit{setcxr}(l, v\,;\mu)$ for $x \in \{a, d\}$.

We have not defined *add1* or *sub1* on anything other than integers nor *car* and *cdr* on $\mathbb{A}$ or $\mathbb{M}^{(n)}_{sexp}$, when $n \neq 1$. We shall not specify their behavior on these sets, the reader should rest assured that the issue is of little importance in this paper.

3. A computation theory over memory structures.

In this section we describe a programming language for computations over memory structures and give this language a semantics. Our language is a first-order lexically scoped Lisp-like language. Although we will work only with the S-expression memory structure, we define the language and semantics for an arbitrary memory structure $\mathbb{M}$ with atoms $\mathbb{A}$, locations $\mathbb{L}$, and operations $\mathbb{O}$. We assume that $\mathbb{A}$ contains a distinguished atom NIL.

3.1. Memory expressions over $\mathbb{M}$.

We begin by defining the expressions of our language. Let $\mathbb{X}$ and $\mathbb{F}$ be disjoint countable sets. Elements of $\mathbb{X}$ are memory variable symbols and range over memory values. Elements of $\mathbb{F}$ are function symbols, each with an associated finite arity. Finally there are constant symbols for the atoms and memory operations of $\mathbb{M}$. However, we will not make any attempt to distinguish between an atom or operation and the constant that denotes it. We use x, $x_0, \ldots$ for elements of $\mathbb{X}$, f, $f_0, \ldots$ for elements of $\mathbb{F}$, and e, $e_0, \ldots$ for memory expressions. The set of *memory expressions* is defined inductively to be the smallest set $\mathbb{E}$ containing

- $\mathbb{V} = \mathbb{A} \oplus \mathbb{L}$
- $\mathbb{X}$

and closed under the following formation rules:

- If $e_{\text{test}}, e_{\text{then}}, e_{\text{else}} \in \mathbb{E}$ then $\texttt{if}(e_{\text{test}}, e_{\text{then}}, e_{\text{else}}) \in \mathbb{E}$.

- If $e_1, \ldots, e_n, e_{body} \in \mathbb{E}$ and $x_1, \ldots, x_n \in \mathbb{X}$ are distinct then

$$\texttt{let}\{x_1 \leftarrow e_1, \ldots, x_m \leftarrow e_m\} e_{body} \in \mathbb{E}.$$

- If $e_1, \ldots, e_n \in \mathbb{E}$ then $\texttt{seq}(e_1, \ldots, e_n) \in \mathbb{E}$.
- If ϑ is either an n-ary memory operation or n-ary function symbol from $\mathbb{F}$, and $e_1, \ldots, e_n \in \mathbb{E}$ then $\vartheta(e_1, \ldots, e_n) \in \mathbb{E}$.

The only variable binding operation is `let`. $\texttt{let}\{y_1 \leftarrow e_1, \ldots, y_m \leftarrow e_m\} e_{body}$ binds the free occurrences of y_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. We say that e is *closed* if $FV(e)$ is empty. $e\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}$ is the result of substituting free occurrences of the y_i in e by the values v_i, or to be more precise the constant symbols denoting them. We often write

$$[e_0, \ldots, e_n]$$

for $\texttt{seq}(e_0, \ldots, e_n)$.

Prior to describing the semantics of memory expressions, we need to make one more definition. A system of memory function definitions is a list of triples

$$\texttt{recdef}((f_0, bs_0, e_0), \ldots, (f_n, bs_n, e_n))$$

that satisfies the following conditions:

- Each bs_i is a sequence, without repetitions, of variables from $\mathbb{X}$ of length m_i.
- f_i is an m_i-ary function symbol from $\mathbb{F}$.
- e_i must be a memory expression such that $FV(e_i)$ is a subset of bs_i, the only function symbols that occur in e_i are among $f_0, \ldots, f_n$, and no $l \in \mathbb{L}$ occurs in any of the e_i.

The `recdef` construct allows us to define a set of mutually recursive functions. The sequence bs_i names the arguments of the function f_i and e_i is the expression used to compute its value. In more traditional notation we have

$$f_0(bs_0) \leftarrow e_0$$

$$\ldots\ldots$$

$$f_n(bs_n) \leftarrow e_b$$

Given such a system of definitions, call it D, we say f_i is defined in D and similar self-explanatory expressions. Note that our language is first order in the sense that we do not have functionals (functions with function parameters or values).

3.2. Rules for computation over memory structures.

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

$$D = \mathtt{recdef}((f_0, bs_0, e_0), \ldots, (f_n, bs_n, e_n)).$$

Then a *memory object description* is pair $e\,;\mu$ that satisfies the following conditions:

- e is a closed memory expression,
- any l that occurs in e is also in δ_μ, and
- every function symbol $f \in \mathbb{F}$ which occurs in e is defined in D.

The basic rules for computation are given by a *single step* relation on memory object descriptions, $e_0\,;\mu_0 \mathrel{-\!\!>}^D e_1\,;\mu_1$, generated by the rules below. That is, $\mathrel{-\!\!>}^D$ is the least relation containing the primitive cases and closed under the congruence conditions. The primitive cases correspond to primitive machine instructions for branching, sequencing, variable binding, execution of memory structure operations and function call. 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.

Primitive cases:

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

$$\mathtt{seq}(e)\,;\mu \mathrel{-\!\!>}^D e\,;\mu$$

$$\mathtt{seq}(v_0, e_1, \ldots, e_m)\,;\mu \mathrel{-\!\!>}^D \mathtt{seq}(e_1, \ldots, e_m)\,;\mu$$

$$\mathtt{let}\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}e\,;\mu \mathrel{-\!\!>}^D e\{y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m\}\,;\mu$$

$$\vartheta(v_1, \ldots, v_n)\,;\mu \mathrel{-\!\!>}^D v_0\,;\mu_0 \qquad \text{if } \vartheta \text{ is a memory operation and } \vartheta(v_1, \ldots, v_n\,;\mu) = v_0\,;\mu_0$$

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

Congruence cases: If $e_a\,;\mu_a \mathrel{-\!\!>}^D e_b\,;\mu_b$ then

$$\mathtt{if}(e_a, e_{\text{then}}, e_{\text{else}})\,;\mu_a \mathrel{-\!\!>}^D \mathtt{if}(e_b, e_{\text{then}}, e_{\text{else}})\,;\mu_b$$

$$\mathtt{seq}(e_a, \ldots)\,;\mu_a \mathrel{-\!\!>}^D \mathtt{seq}(e_b, \ldots)\,;\mu_b$$

$$\mathtt{let}\{y_1 \leftarrow v_1, \ldots, y_{j-1} \leftarrow v_{j-1}, y_j \leftarrow e_a, \ldots, y_m \leftarrow e_m\}e\,;\mu_a \mathrel{-\!\!>}^D$$
$$\mathtt{let}\{y_1 \leftarrow v_1, \ldots, y_{j-1} \leftarrow v_{j-1}, y_j \leftarrow e_b, \ldots, y_m \leftarrow e_m\}e\,;\mu_b$$

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

The ***reduction relation*** on memory object descriptions, $e_0 ; \mu_0 \gg^D e_1 ; \mu_1$, is the transitive closure of the single step relation. We say $e ; \mu$ *evaluates* to $v_0 ; \mu_0$ if $e ; \mu \gg^D v_0 ; \mu_0$ for some $v_0 \in \mathbb{V}$. We can now easily describe the functions defined by our `recdef` construct. Namely if ϑ is defined in D and $(y_1, \ldots, y_n)$ is its binding specification then the corresponding partial function

$$\vartheta^D : \mathbb{M}^{(n)} \rightharpoonup \mathbb{M}$$

is defined by

$$\vartheta^D(v_1, \ldots, v_n ; \mu) \rightharpoonup v_0 ; \mu_0 \underset{\text{df}}{\equiv} \vartheta(v_1, \ldots, v_n) ; \mu \gg^D v_0 ; \mu_0$$

In the following we generally work with a fixed D and will omit the definition superscript on the reduction relation.

3.3. Remarks

• 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 $e ; \mu_0 \gg v ; \mu_1$.

• 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, $car(l;\mu)$ is a term and $car(l);\mu$ is a memory object description, and we have $car(l) ; \mu \gg car(l ; \mu)$. The two uses of operation and function symbols should cause no confusion.

• The values of the binding expressions of a `let` construct are evaluated in sequence. Then the free occurences of the variables in the body of the expression are replaced by the corresponding values. The binding expressions are evaluated in their original environment and not the one being created by the `let` . The `seq` construct provides for sequencing of computations. It is similar to the PROGN construct of Lisp. We should point out that `seq` is definable in terms of `let` since

$$\texttt{seq}(e_0, e_1, \ldots, e_n)$$

is equivalent to

$$\texttt{let}\ \{x_0 \leftarrow e_0, x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\} x_n$$

Definition by cases is handled by the `if` construct. Notice that as usual in Lisp any non-`NIL` value of the test is considered *true*.

• We have not included a means of dynamically assigning values to variables, such as the Lisp SETQ mechanism. For present purposes the inclusion of such mechanisms mainly complicates the semantics. They become interesting in a computation theory where functions can be returned as values.

• Our notion of memory structure is essentially that of Burstall [B], although the presentations are somewhat different. Burstall treats computations described by flowchart programs and develops proof rules for proving properities of certain list and tree like memories. We treat computations described by systems of recursive definition and prove properties of the functions described by these computations. In this paper we treat a larger variety of programs acting on much less restricted domains. We focus on mathematical properties of the S-expression domain and do not develop any formal proof-rules.

3.4. Abbreviations

In addition to the basic constructs of our language, we also use constructs like and, not, or and ifn. They are taken to be the usual Lisp abbreviations or *macros* namely:

$$\mathtt{and}(e_1, e_2) \underset{\text{df}}{=} \mathtt{if}(e_1, e_2, \mathtt{NIL})$$

$$\mathtt{or}(e_1, e_2) \underset{\text{df}}{=} \mathtt{if}(e_1, \mathtt{T}, e_2)$$

$$\mathtt{not}(e) \underset{\text{df}}{=} \mathtt{if}(e, \mathtt{NIL}, \mathtt{T})$$

$$\mathtt{ifn}(e_{\text{test}}, e_{\text{then}}, e_{\text{else}}) \underset{\text{df}}{=} \mathtt{if}(e_{\text{test}}, e_{\text{else}}, e_{\text{then}})$$

In addition we have a *cond*-like construct ifs, where

$$\mathtt{ifs}(e^0_{\text{test}}, e^0_{\text{then}} \cdots e^n_{\text{test}}, e^n_{\text{then}}) \underset{\text{df}}{=} \mathtt{if}(e^0_{\text{test}}, e^0_{\text{then}}, \mathtt{if}(e^1_{\text{test}}, e^1_{\text{then}} \cdots \mathtt{if}(e^n_{\text{test}}, e^n_{\text{then}}, \mathtt{NIL}) \ldots))).$$

It is common in Lisp programs to test for atoms rather than pairs. The test *atom* is defined by

$$\mathit{atom}(e) \underset{\text{df}}{=} \mathtt{not}(\mathit{cons?}(e)).$$

Rather than explicitly use the recdef function we write function definitions in the traditional manner, only implicitly using the recdef operator. For example the definitions of *append* and *memq* are

$$\mathit{append}(\mathtt{u},\mathtt{v}) \leftarrow \mathtt{if}(\mathtt{u}, \mathit{cons}(\mathit{car}(\mathtt{u}), \mathit{append}(\mathit{cdr}(\mathtt{u})), \mathtt{v})), \mathtt{v})$$

$$\begin{aligned}
&\mathit{memq}(\mathtt{element}, \mathtt{list}) \leftarrow \\
&\quad \mathtt{if}(\mathtt{list}, \\
&\qquad \mathtt{if}(\mathit{eq}(\mathtt{element}, \mathit{car}(\mathtt{list})), \\
&\qquad\quad \mathtt{T}, \\
&\qquad\quad \mathit{memq}(\mathtt{element}, \mathit{cdr}(\mathtt{list}))), \\
&\qquad \mathtt{NIL})
\end{aligned}$$

We are also somewhat liberal in what we shall use as variables, using words with suggestive names. If D is the system of definitions

$$\mathtt{recdef}((f_0, bs_0, e_0), \ldots, (f_n, bs_n, e_n)),$$

then we say D is a *tail-recursive* system if and only if no function symbol f_i, which is defined in D, appears in D either in:

1. The test-expression of an `if` expression,
2. a binding expression of a `let` expression,
3. an expression other than the last in a `seq` , or
4. an expression that is an argument to a function or operation symbol in D.

It is well known that functions so defined can be implemented on low-level machines without the use of a stack, see for example [**Tu**] or [**F**]. For example, the following definition of the list length function

$$length(\mathtt{l}) \leftarrow \mathtt{if}(\mathtt{l}, add1\,(length(cdr(\mathtt{l}))), \mathtt{0})$$

is not tail-recursive. Whereas the following system, which defines an ***extensionally*** equivalent function, is tail-recursive.

$$length(\mathtt{l}) \leftarrow len(\mathtt{l}, \mathtt{0})$$
$$len(\mathtt{l}, \mathtt{n}) \leftarrow \mathtt{if}(\mathtt{l}, len(cdr(\mathtt{l}), add1\,(\mathtt{n})), \mathtt{n})$$

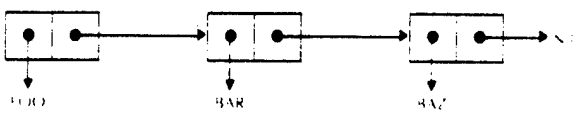
4. More about $\mathbb{M}_{sexp}$.

In this section we study the particular memory structure $\mathbb{M}_{sexp}$ that we defined in section 2. It will be the principle memory structure that we shall deal with in the rest of this paper. Henceforth all memory objects will be assumed to be in $\mathbb{M}_{sexp}$ unless otherwise stated. Hopefully by the end of this section any person that has used Lisp or has been subjected to a mathematical treatment of Lisp-like languages will have developed a practical intuition about this memory structure model. We begin by showing how a memory object $v\,;\mu \in \mathbb{M}_{sexp}$ can be represented using the traditional Lisp ***boxes and pointers*** notation.

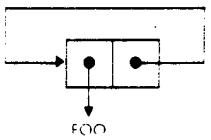
For example if we let μ_0 be the memory:

$$\{< l_0, [\mathtt{FOO}, l_1] >, < l_1, [\mathtt{BAR}, l_2] >, < l_2, [\mathtt{BAZ}, \mathtt{NIL}] >\}$$

then we can represent the memory object $l_0\,;\mu_0$ by the following diagram (which is taken from [**To**]):



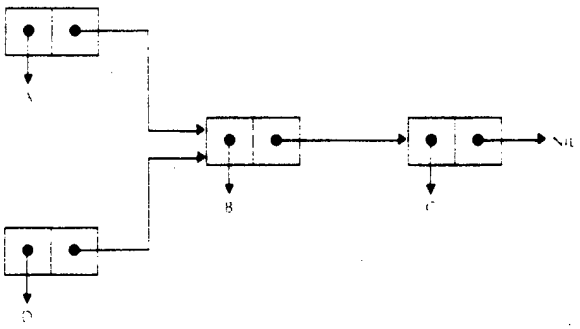
If $rplacd(l_0, l_0)\,;\mu_0 \gg l_0\,;\mu_1$ then the memory object $l_0\,;\mu_1$ looks like



Another example where two memory objects share structure is in μ_2, where μ_2 is:

$$\{< l_0, [\mathtt{A}, l_2] >, < l_1, [\mathtt{D}, l_2] >, < l_2, [\mathtt{B}, l_3] >, < l_3, [\mathtt{C},\ \mathtt{NIL}] >\}$$

μ_2 itself can be represented by the diagram:



We often use the suggestive boxes and pointers way of speaking about memory objects when it suits our purpose. The boxes we call *cells* and a pointer is just another way of refering to the a location or *cell*. Henceforth *cell*, *location*, and *label* will be used synonymously.

4.1. Viewing memory objects as labeled trees

There is a very simple way of regarding an S-expression memory object as a labelled tree. For $v\,;\mu \in \mathbb{M}_{sexp}$ we define a partial function $\lambda x.(v\,;\mu)_x$ from $\mathbb{T}$ to $\mathbb{V}$ and its domain $\delta_{\lambda x.(v;\mu)_x}$ by induction on $\mathbb{T}$:

$$(v\,;\mu)_\sigma = \begin{cases} v & \text{if } \sigma = \square, \text{ the empty word in } \mathbb{T} \\ \mu((v\,;\mu)_{\sigma_0})\downarrow_i & \text{if } \sigma = \sigma_0 \diamond i,\ i \in 2 \text{ and } (v\,;\mu)_{\sigma_0} \in \mathbb{L} \end{cases}$$

When refering to tree function $\lambda x.(v\,;\mu)_x$ we generally drop the λ and simply write $(v\,;\mu)$. Thus, $(v\,;\mu)$ is the least function from $\mathbb{T}$ to $\mathbb{V}$ satisfying:

- $\square \in \delta_{(v;\mu)}$ and $(v\,;\mu)_\square = v$

and if $\sigma \in \delta_{(v;\mu)}$ and $(v\,;\mu)_\sigma = l \in \mathbb{L}$ then

- $\sigma \diamond j \in \delta_{(v;\mu)}$, and
- $(v\,;\mu)_{\sigma\diamond j} = \mu(l)\downarrow_j$ for $j \in 2$

Our notation in this regard is similar to that of [Mo].

We call $(v\,;\mu)$ the *derived tree function*, or the *labelled tree* that is defined by $v\,;\mu$. Note that the following facts are true for these functions.

Proposition 1: For any $v\,;\mu$, $\delta_{(v;\mu)}$ is a non-empty subtree of $\mathbb{T}$, with the property that if $\sigma \diamond j \in \delta_{(v;\mu)}$ for $j \in 2$ then $(v\,;\mu)_\sigma \in \mathbb{L}$. If $\sigma_0 \diamond \sigma_1 \in \delta_{(v;\mu)}$ then

1. $\sigma_0 \in \delta_{(v;\mu)}$ and $\sigma_1 \in \delta_{((v;\mu)_{\sigma_0};\mu)}$
2. $(v\,;\mu)_{\sigma_0\diamond\sigma_1} = ((v\,;\mu)_{\sigma_0}\,;\mu)_{\sigma_1}$

***nthcdr* example:** Consider the following well-known Lisp program

$$nthcdr(\mathtt{n},\mathtt{l}) \leftarrow \mathtt{if}(eq(\mathtt{n},\mathtt{0}),\mathtt{l},nthcdr(sub1(\mathtt{n}),cdr(\mathtt{l})))$$

The significance of this function is expressed by

$$nthcdr(n,l) = (l\,;\mu)_{1^n}\,;\mu$$

when both sides are defined, or equivalently when either is defined.

We will sometimes refer to σ (when σ is in the domain of the derived tree function of a memory object $v\,;\mu$) as a *car-cdr* chain in $v\,;\mu$, for the obvious reason that $(v\,;\mu)_\sigma$ is the location or cell one obtains by a suitable composition of the memory operation *car* and *cdr*. Thus we can define the notion of the cells of a memory object which are accessible by *car-cdr* chains.

We define $\mathbf{Cells}_\mu(v)$ to be set of cells that are reachable from $v\,;\mu$ by travelling along any *car-cdr* chain, and $\mathbf{Cells}^<_\mu(v)$ to be set of cells reachable from $v\,;\mu$ by travelling along any non-empty *car-cdr* chains. Thus

$$\mathbf{Cells}_\mu(v) \underset{\text{df}}{=} \{l \in \mathbb{L} \mid (\exists\sigma)(v\,;\mu)_\sigma = l\}$$

$$\mathbf{Cells}^<_\mu(v) \underset{\text{df}}{=} \{l \in \mathbb{L} \mid (\exists\sigma \neq \square)(v\,;\mu)_\sigma = l\}$$

Notice that we could also define $\mathbf{Cells}_\mu(v)$ to be the smallest subset X of δ_μ such that by letting μ_X be the restriction of μ to X, we have

$$v\,;\mu_X \in \mathbb{M}_{sexp}.$$

Consequently, if we were only interested in the memory object $v;\mu$ it would for most intents and purposes be reasonable to assume that $\mathbf{Cells}_\mu(v) = \delta_\mu$.

We often wish to define a set of cells (or values) that have a particular property and are reachable from a given cell via paths which only pass through cells with this property. The following constructions give a general way of making this type of definition. Let Ψ, Φ_i be predicates on $\mathbb{M}_{sexp}$, for $i \in 2$, then

- $\mathbf{TC}(v\,;\mu, \Psi, \Phi_0, \Phi_1)$ to be the smallest set X such that
  1. If $\Psi(v\,;\mu)$ then $v \in X$
  2. If $l \in X$ and $\Phi_0(l\,;\mu)$ then $(l\,;\mu)_0 \in X$
  3. If $l \in X$ and $\Phi_1(l\,;\mu)$ then $(l\,;\mu)_1 \in X$
- $\mathbf{STC}(v\,;\mu, \Phi_0, \Phi_1)$ is the smallest set X such that
  1. If $l \in X$ or $l = v$ then if $\Phi_0(l\,;\mu)$ then $(l\,;\mu)_0 \in X$
  2. If $l \in X$ or $l = v$ then if $\Phi_1(l\,;\mu)$ then $(l\,;\mu)_1 \in X$

For example

$$\mathbf{Cells}_\mu(v) = \mathbf{TC}(v\,;\mu, \Psi_{\mathbb{L}}, \Phi^0_{\mathbb{L}}, \Phi^1_{\mathbb{L}}) \quad \text{and} \quad \mathbf{Cells}^<_\mu(v) = \mathbf{STC}(v\,;\mu, \Phi^0_{\mathbb{L}}, \Phi^1_{\mathbb{L}})$$

where $v\,;\mu \in \Phi^i_{\mathbb{L}}$ iff $v \in \mathbb{L}$ and $(\mu\,;v)_i \in \mathbb{L}$ and $v\,;\mu \in \Psi_{\mathbb{L}}$ iff $v \in \mathbb{L}$. We shall make frequent use of these constructions in later sections. We sometimes write $\mathbf{TC}(v\,;\mu, \Phi)$ or $\mathbf{STC}(v\,;\mu, \Phi)$ when all three predicates are the same. **TC** stands for transitive closure, while **STC** stands for strict transitive closure.

4.2. Equivalence relations on memory objects

Often a memory structure contains more detail than is necessary for the task at hand, for this reason we define two notions of similarity. The first is the most obvious. We say two memories μ_0 and μ_1 are isomorphic, written $\mu_0 \cong \mu_1$, if there is a bijection, h, from $\mathbb{V}$ to $\mathbb{V}$ which is the identity on $\mathbb{A}$ and maps $\mathbb{L}$ to $\mathbb{L}$ with the property that $h \circ \mu_0 = \mu_1$. Since we mainly deal with memory objects not simply just memories, we also define the corresponding notion for memory objects. For this we use the tree-function $(v;\mu)$ associated with $v\,;\mu$.

Definition of isomorphic memory objects: If $v_0, \ldots, v_n\,;\mu,\ v^*_0, \ldots, v^*_n\,;\mu^* \in \mathbb{M}^{(n+1)}_{sexp}$ 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 $h\colon \mathbb{V} \to \mathbb{V}$ which is the identity on $\mathbb{A}$, maps $\mathbb{L} \to \mathbb{L}$ and is such that

$$h \circ (v_i\,;\mu) = (v^*_i\,;\mu^*)$$

as partial functions, for every $i \in n+1$.

Notice that if $\mathbf{Cells}_{\mu_i}(v_i) = \delta_{\mu_i}$, for $i \in 2$, then saying that $v_0\,;\mu_0 \cong v_1\,;\mu_1$ is the same as saying that $\mu_0 \cong \mu_1$ via h where h has the additional property that $h(v_0) = v_1$. Another

important point to observe is that S-expression memory operations preserve isomorphism. For example,

$$l, v\ ;\mu \cong l^*, v^*\ ;\mu^* \ \rightarrow\ rplaca(l, v\ ;\mu) \cong rplaca(l^*, v^*\ ;\mu^*).$$

Note that

$$v_0\ ;\mu \cong v_0^*\ ;\mu^* \wedge \ldots \wedge v_n\ ;\mu \cong v_n^*\ ;\mu^* \not\rightarrow v_0, \ldots, v_n\ ;\mu \cong v_0^*, \ldots, v_n^*\ ;\mu^*.$$

Another equivalence relation that is not quite as useful in this paper, but does have a special significance in the subject is that of Lisp equality.

Definition of Lisp equality: We say $v_0\ ;\mu_0$ and $v_1\ ;\mu_1$ are Lisp equal, written

$$v_0\ ;\mu_0 \equiv v_1\ ;\mu_1,$$

iff $(v_0\ ;\mu_0)$ and $(v_1\ ;\mu_1)$ have the same domains and $(v_0\ ;\mu_0)_\sigma = a$ when and only when $(v_1\ ;\mu_1)_\sigma = a$, for $\sigma \in \delta_{(v;\mu_1)}$, $a \in \mathbb{A}$.

Notice that $v_0\ ;\mu_0 \equiv v_1\ ;\mu_1$ means that $v_0\ ;\mu_0$ and $v_1\ ;\mu_1$ have exactly the same *car-cdr* chains. Also, Lisp equal objects ***print*** the same (for typical printing algorithms). As we have already mentioned:

Proposition 2:

1. $\equiv$ and $\cong$ are both equivalence relations.
2. If $v_0\ ;\mu_0 \cong v_1\ ;\mu_1$ then $v_0\ ;\mu_0 \equiv v_1\ ;\mu_1$, the converse is patently false.
3. If D is a definition 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.

We should also point out that more model theoretic definitions of these two equivalence relations are possible, but we shall not do this here. For $v_0, v_1 \in \mathbb{V}$ we say $v_0 \equiv v_1$ iff either v_0 and $v_1 \in \mathbb{L}$ or else $v_0 = v_1$. Using this we have the following pointwise characterization of $\equiv$.

Proposition 3: The following are equivalent

1. $v_0\ ;\mu_0 \equiv v_1\ ;\mu_1$
2. $\delta_{(v_0;\mu_0)} = \delta_{(v_1;\mu_1)} = \gamma$ and $\forall \sigma \in \gamma\ (v_0\ ;\mu_0)_\sigma \equiv (v_1\ ;\mu_1)_\sigma$.

Notice that proposition 1 together with proposition 3. implies

Proposition 4: If $l_0\ ;\mu$ and $l_1\ ;\mu \in \mathbb{M}_{sexp}$ then the following are equivalent

1. $l_0\ ;\mu \equiv l_1\ ;\mu$
2. $(l_0\ ;\mu)_i\ ;\mu \equiv (l_1\ ;\mu)_i\ ;\mu$ for $i \in 2$.

In other words two S-expressions are Lisp equal iff their *car*s and *cdr*s are.

4.3. Some sub-domains of $\mathbb{M}_{sexp}$

We now define some important subdomains of $\mathbb{M}_{sexp}$, and terminology that we use correspondingly.

Definition of well-founded S-expressions: We say that $v\,;\mu$ is a well-founded S-expression, written $v;\mu \in \mathbb{M}_{wfsexp}$, if $\delta_{(v;\mu)}$ is a well-founded tree. Here are several equivalent ways of expressing well-foundedness. One is

$$v \notin \mathbf{Cells}^{<}_{\mu}(v) \ \wedge\ (\forall l \in \mathbf{Cells}^{<}_{\mu}(v))(l \notin \mathbf{Cells}^{<}_{\mu}(l)).$$

Thus if $l;\mu \in \mathbb{M}_{wfsexp}$ then all *car-cdr* chains in $l;\mu$ must eventually terminate at an element of $\mathbb{A}$. A second equivalent definition is that the derived labelled tree is finite. It is important to notice that if $l^* \in \mathbf{Cells}_{\mu}(l)$ and $l\,;\mu \in \mathbb{M}_{wfsexp}$ then

$$\mathbf{Cells}_{\mu}(l^*) \subseteq \mathbf{Cells}_{\mu}(l)$$

with equality holding only when $l^* = l$. Also notice that when $l\,;\mu \in \mathbb{M}_{wfsexp}$ then

$$\mathbf{Cells}_{\mu}(l) = \mathbf{Cells}^{<}_{\mu}(l) \cup \{l\}$$

and this union is disjoint, while disjointness is not necessarily true if we only know that $l\,;\mu \in \mathbb{M}_{sexp}$. We make two last remarks concerning $\mathbb{M}_{wfsexp}$. $\mathbb{M}_{wfsexp}$ factored out by $\equiv$ is canonically isomorphic to the structure one obtains by closing $\mathbb{A}$ under a pairing operation, see for example [**Mo**]. Secondly, for any memory object $v\,;\mu \in \mathbb{M}_{wfsexp}$ there is a closed term e, i.e one with no free variables, which contains only the operations *car*, *cdr* and *cons*, and of course no function symbols, such that $e\,;\emptyset \gg v^*\,;\mu^*$ and $v\,;\mu \cong v^*\,;\mu^*$. Here $\emptyset$ denotes the empty memory. If we do not include the `let` construct in the set of terms, then we can only obtain $\equiv$ in this last result.

Definition of lists: There are two different notions of list depending on whether one allows cyclic lists, in this paper we will refer to the non-cyclic version as $\mathbb{M}_{list}$ and the possibly infinite variety by $\mathbb{M}_{elist}$.

$$v\,;\mu \in \mathbb{M}_{list} \leftrightarrow (\exists n \in \mathbb{N})(v\,;\mu)_{1^n} = \mathtt{NIL}.$$

Thus $l\,;\mu$ is in $\mathbb{M}_{list}$ iff some *cdr*-chain leads to an atom and this atom is `NIL`.

$$v\,;\mu \in \mathbb{M}_{elist} \leftrightarrow (\forall n \in \mathbb{N})(1^n \in \delta_{(v;\mu)} \ \wedge\ (v\,;\mu)_{1^n} \in \mathbb{A} \rightarrow (v\,;\mu)_{1^n} = \mathtt{NIL}).$$

A simple example of a function on $\mathbb{M}_{list}$ is *length*, (defined in section 3.4). Its basic property is that for any $v\,;\mu \in \mathbb{M}_{list}$ we have that $(v\,;\mu)_{1^{length(v)}} = \mathtt{NIL}$. Later on we will describe a length function that is defined for all of $\mathbb{M}_{elist}$. To make talking about lists somewhat easier we have the following notation. The set of cells that are reachable from a non-`NIL` elist $l\,;\mu \in \mathbb{M}_{elist}$ only by using the the function *cdr* is called the *spine* of the list. Namely

$$\mathbf{Spine}_{\mu}(l) = \{(l\,;\mu)_{1^n} \mid 1^n \in \delta_{(l;\mu)}\} - \{\mathtt{NIL}\}.$$

Suppose $l_0 ; \mu_0 \in \mathbb{M}_{list}$ is such that

$$\mathbf{Spine}_{\mu_0}(l_0) = \{l_0 \ldots l_n\}$$

with $\mu_0(l_i) = [v_i \ , \ l_{i+1}]$ for $i \in n$ and $\mu_0(l_n) = [v_n \ , \ \mathtt{NIL}]$. Then we say $l_0 ; \mu_0$ represents the Lisp list $(v_0 \ v_1 \ v_2 \ \ldots \ v_n)$. We call the v_i the elements of the list $l_0 ; \mu_0$ and put $\mathbf{Elements}_{\mu_0}(l_0) = [v_0, \ldots, v_n]$. We say $l_0 ; \mu_0$ is a *pure* list if $\mathbf{Spine}_{\mu_0}(l_0)$ is disjoint from the set

$$\bigcup_{v_i \in \mathbf{Elements}_{\mu_0}(l_0)} \mathbf{Cells}_{\mu_0}(v_i).$$

Thus a pure list is determined up to isomorphism by the sequence of its elements.

4.4. The Equality Program

We finish of this section by showing that our notion of Lisp equality agrees with the usual notion on $\mathbb{M}_{wfsexp}$. Consider the following well known program.

$$\begin{array}{l} equal(\mathrm{u},\mathrm{v}) \leftarrow \\ \quad \mathtt{if}(\mathtt{or}(atom(\mathrm{u}), atom(\mathrm{v})), \\ \quad\quad eq(\mathrm{u},\mathrm{v}), \\ \quad\quad \mathtt{and}(equal(car(\mathrm{u}), car(\mathrm{v})), \\ \quad\quad\quad equal(cdr(\mathrm{u}), cdr(\mathrm{v})))) \end{array}$$

Theorem 1: *equal* is a total function from $\mathbb{M}^{(2)}_{wfsexp}$ to $\mathbb{M}_{wfsexp}$, having values amongst $\{\mathtt{NIL}, \mathtt{T}\}$. Further, if $v_0 ; \mu, \ v_1 ; \mu \in \mathbb{M}_{wfsexp}$ then the following are equivalent:

1. $equal(v_0, v_1) ; \mu \gg \mathtt{T} ; \mu$
2. $v_0 ; \mu \equiv v_1 ; \mu$

Proof: We prove the theorem by induction on

$$r(v_0, v_1 ; \mu) = |\mathbf{Cells}_{\mu}(v_0)| \times |\mathbf{Cells}_{\mu}(v_1)|.$$

Base case: $r(v_0, v_1 ; \mu) = 0$. In this case $v_i \in \mathbb{A}$ for at least one $i \in 2$, and so

$$equal(v_0, v_1) ; \mu \gg eq(v_0, v_1) ; \mu.$$

Since we have that $eq(v_0, v_1) ; \mu \gg \mathtt{T} ; \mu$ iff $v_0 = v_1$ and $v_0 ; \mu \equiv v_1 ; \mu$ iff $v_0 = v_1$ the theorem is true in this case.

Induction step: Suppose $r(v_0, v_1 ; \mu) > 0$ and that the theorem is true for any $v_2, v_3 ; \mu_0 \in \mathbb{M}_{wfsexp}$ of less rank. Thus v_0 and $v_1 \in \mathbb{L}$ and

$$equal(v_0, v_1) ; \mu \gg \mathtt{and}(equal(car(v_0), car(v_1)), equal(cdr(v_0), cdr(v_1))) ; \mu$$

If we let $v_{ia} = \mu(v_i)\downarrow_0$ and $v_{id} = \mu(v_i)\downarrow_1$, for $i \in 2$ then we have

$$\mathit{equal}(v_0, v_1)\ ;\mu \gg \mathrm{and}(\mathit{equal}(v_{0a}, v_{1a}), \mathit{equal}(\mathit{cdr}(v_0), \mathit{cdr}(v_1)))\ ;\mu.$$

Now since $v_i \in \mathbb{M}_{wfsexp}$ we have that $r(v_{0a}, v_{1a}\ ;\mu), r(v_{0d}, v_{1d}\ ;\mu) < r(v_0, v_1\ ;\mu)$. Consider two cases.

Case 1: If $v_0\ ;\mu \equiv v_1\ ;\mu$ then by proposition 4. $v_{0a}\ ;\mu \equiv v_{1a}\ ;\mu$ and $v_{0d}\ ;\mu \equiv v_{1d}\ ;\mu$. So

$$\mathit{equal}(v_0, v_1)\ ;\mu \gg \mathrm{and}(\ \mathtt{T}, \mathit{equal}(\mathit{cdr}(v_0), \mathit{cdr}(v_1)))\ ;\mu \gg \mathit{equal}(v_{0d}, v_{1d})\ ;\mu \gg \mathtt{T}\ ;\mu.$$

Case 2: If $v_0\ ;\mu \not\equiv v_1\ ;\mu$ then again by proposition 4 either $v_{0a}\ ;\mu \not\equiv v_{1a}\ ;\mu$ or $v_{0d}\ ;\mu \not\equiv v_{1d}\ ;\mu$. Suppose $v_{0a}\ ;\mu \not\equiv v_{1a}\ ;\mu$ then

$$\mathit{equal}(v_0, v_1)\ ;\mu \gg \mathrm{and}(\mathtt{NIL}, \mathit{equal}(\mathit{cdr}(v_0), \mathit{cdr}(v_1)))\ ;\mu.$$

However, if $v_{0a}\ ;\mu \equiv v_{1a}\ ;\mu$, then

$$\mathit{equal}(v_0, v_1)\ ;\mu \gg \mathit{equal}(\mathit{cdr}(v_0), \mathit{cdr}(v_1))\ ;\mu \gg \mathtt{NIL}\ ;\mu.$$

□Theorem 1

One final remark is that the above proof can easily be modified to show that the more efficient version of equal given below also satisfies this theorem.

```
equal(u,v) ←
    if(eq(u,v),T,if(or(atom(u),atom(v)),
                    NIL,
                    and(equal(car(u),car(v)),
                        equal(cdr(u),cdr(v)))))
```

5. Four simple correctness proofs.

In this section we present four well known Lisp programs, and prove theorems asserting their correctness. None of the proofs is in any way deep, the main purpose being tutorial, in that we show both how to formulate correctness results and how they are proved. The reader who is not so interested in methodology but rather results should simply skip the proofs, as the specific techniques of proof in this section are not duplicated in the subsequent, more complex proofs.

5.1. Example 1: Inplace Reverse.

In this example we prove the correctness of a destructive reverse program, the so called *inplace* reverse.

$$\begin{aligned}
&\textit{inplace:reverse}(\mathtt{u}) \leftarrow \textit{in:rev}(\mathtt{u},\mathtt{NIL})\\
&\textit{in:rev}(\mathtt{u},\mathtt{v}) \leftarrow\\
&\quad \mathtt{if}(\mathtt{u},\\
&\qquad \mathtt{let}\{\mathtt{t1} \leftarrow \textit{cdr}(\mathtt{u})\}\mathtt{seq}(\textit{rplacd}(\mathtt{u},\mathtt{v}),\textit{in:rev}(\mathtt{t1},\mathtt{u})),\\
&\qquad \mathtt{v}))
\end{aligned}$$

Clearly $\textit{inplace:reverse}(\mathtt{NIL})\ ;\mu \gg \mathtt{NIL}\ ;\mu$. In general *inplace:reverse* reverses a list by reversing the pointers along the spine and changing nothing else. This is expressed by

Theorem 2: If $l_0 ;\mu_0 \in \mathbb{M}_{list}$ represents the Lisp list $(v_0\ v_1\ v_2\ \ldots\ v_n)$ with $\mathbf{Spine}_{\mu_0}(l_0) = \{l_0 \ldots l_n\}$ then

$$\textit{inplace:reverse}(l_0)\ ;\mu_0 \gg l_n\ ;\mu_{n+1}$$

where $l_n\ ;\mu_{n+1}$ represents the Lisp list $(v_n\ v_{n-1}\ \ldots\ v_2\ v_1\ v_0)$, $\mathbf{Spine}_{\mu_{n+1}}(l_n) = \{l_n \ldots l_0\}$, $\mu_1 = \textit{setcdr}(l_0,\ \mathtt{NIL}\ ;\ \mu_0)$, and $\mu_{i+1} = \textit{setcdr}(l_i,\ l_{i-1}\ ;\ \mu_i)$, for $i \in n+1$. In addition $\delta_{\mu_0} = \delta_{\mu_{n+1}}$ with μ_{n+1} differing from μ_0 only on $\{l_i\}_{i \in n+1}$.

Corollary 1: $\textit{inplace:reverse}(\textit{inplace:reverse}(l_0\ ;\mu_0)) = l_0\ ;\mu_0$

Notice that unless $l_0\ ;\mu_0$ is a pure list we will not have that $v_i\ ;\mu_{n+1} \equiv v_i\ ;\mu_0$, in other words *inplace:reverse* may alter the elements of the original list. However a little careful thought on the matter will show that there is no particularly obvious candidate for the epitaph *reverse* of a list in such structure sharing situations.

Proof of Theorem: We will show by induction on i that

P1. $\textit{in:rev}(l_0,\mathtt{NIL})\ ;\mu_0 \gg \textit{in:rev}(l_{i+1},l_i)\ ;\mu_{i+1} \gg \textit{in:rev}(\mathtt{NIL},l_n)\ ;\mu_{n+1}$

P2. $i < j \leq n \rightarrow \mu_0(l_j) = \mu_{i+1}(l_j)$

P3. $0 \leq j < i \rightarrow \mu_{i+1}(l_j) = \mu_{j+1}(l_j)$

Note that

- for $0 < j \leq n$ $\mu_0(l_j) = [v_j,l_{j+1}]$ and $\mu_{j+1}(l_j) = [v_j,l_{j-1}]$
- for any $l\ ;\mu \in \mathbb{M}_{list}$ with $u\ ;\mu = cdr(l)\ ;\mu$ we have by computation

$$\begin{aligned}
&\textit{in:rev}(l,v)\ ;\mu \gg\\
&\quad \gg \mathtt{if}(l,\mathtt{let}\{t_1 \leftarrow cdr(l)\}[\textit{rplacd}(l,v),\textit{in:rev}(t_1,l)],v)\ ;\mu\\
&\quad \gg \mathtt{let}\{t_1 \leftarrow cdr(l)\}[\textit{rplacd}(l,v),\textit{in:rev}(t_1,l)]\ ;\mu\\
&\quad \gg \textit{in:rev}(u,l)\ ;\textit{setcdr}(l,v\ ;\mu)
\end{aligned}$$

- since $l_0 ; \mu_0 \in \mathbb{M}_{list}$ we have $l_i \neq l_j$, whenever $i \neq j$, and $i,j \in n+1$.

Case $i = 0$: By computation, since $l_1 ; \mu_0 = cdr(l_0) ; \mu_0$ and $\mu_1 = setcdr(l_0, \mathtt{NIL} ; \mu_0)$ we have

$$in{:}rev(l_0, \mathtt{NIL}) ; \mu_0 \gg in{:}rev(l_1, l_0) ; \mu_1$$

Thus P1 holds for $i = 0$. Since μ_1 differs from μ_0 only on l_0 we have that

$$\mu_0(l_s) = \mu_1(l_s) \text{ for } 0 < s \leq n$$

so P2 holds. P3 is vacuous.

Induction step: Suppose $0 < i < n$ and

$$in{:}rev(l_0, \mathtt{NIL}) ; \mu_0 \gg in{:}rev(l_i, l_{i-1}) ; \mu_i$$

with μ_j satisfying P2 for $i-1 \leq j \leq n$ and P3 for $0 \leq j < i-1$. Thus $l_{i+1} ; \mu_i = cdr(l_i) ; \mu_0 = cdr(l_i) ; \mu_i$. By computation again we have

$$in{:}rev(l_i, l_{i-1}) ; \mu_i \gg in{:}rev(l_{i+1}, l_i) ; \mu_{i+1}$$

where $\mu_{i+1} = setcdr(l_i, l_{i-1} ; \mu_i)$. P2 and P3 hold for μ_{i+1} because it only differs from μ_i on l_i.

Termination case: So far we have shown that for $0 \leq i \leq n$

$$in{:}rev(l_0, \mathtt{NIL}) ; \mu_0 \gg in{:}rev(l_i, l_{i-1}) ; \mu_i$$

with μ_j satisfying P2 for $i \leq j \leq n$ and P3 for $0 \leq j < i$. Thus P2 and P3 are proved and $cdr(l_n) ; \mu_n = \mathtt{NIL} ; \mu_n$. By computation we have

$$in{:}rev(l_n, l_{n-1}) ; \mu_n \gg in{:}rev(\mathtt{NIL}, l_n) ; \mu_{n+1}$$

where $\mu_{n+1} = setcdr(l_n, l_{n-1} ; \mu_n)$.

$\square_{P1,P2,P3}$

The theorem now follows from the above and the simple observation that

$$inplace{:}reverse(l_0) ; \mu_0 \gg in{:}rev(l_0, \mathtt{NIL}) ; \mu_0 \gg l_n ; \mu_{n+1}.$$

$\square_{\textbf{Theorem 2}}$

5.2. Example 2: Iterative Append.

We now prove the correctness of an iterative append program. It constructs a list with the same elements as its first argument and destructively appends the second argument to the end of this new list. It does not alter the original memory on any pre-existing location, thus it is sometimes called a *locally dirty* program.

$$
\begin{array}{l}
\mathit{iterative{:}append}(\mathtt{u},\mathtt{v}) \leftarrow \\
\quad \mathtt{if}(\mathtt{u}, \\
\quad\quad \mathtt{v}, \\
\quad\quad \mathtt{let}\{\mathtt{w} \leftarrow \mathit{cons}(\mathit{car}(\mathtt{u}),\mathtt{v})\}\, \mathit{it{:}app}(\mathtt{v},\mathtt{w},\mathtt{w},\mathit{cdr}(\mathtt{u}))) \\
\mathit{it{:}app}(\mathtt{v},\mathtt{val},\mathtt{w},\mathtt{u}) \leftarrow \\
\quad \mathtt{if}(\mathtt{u}, \\
\quad\quad \mathtt{val}, \\
\quad\quad \mathit{it{:}app}(\mathtt{v}, \\
\quad\quad\quad\quad \mathtt{val}, \\
\quad\quad\quad\quad \mathit{cdr}(\mathit{rplacd}(\mathtt{w},\mathit{cons}(\mathit{car}(\mathtt{u}),\mathtt{v}))), \\
\quad\quad\quad\quad \mathit{cdr}(\mathtt{u})))
\end{array}
$$

Clearly $\mathit{iterative{:}append}(\mathtt{NIL}, v)\,;\mu \gg v\,;\mu$.

Theorem 3: If $l_0\,;\mu_0 \in \mathbb{M}_{list}$ represents the Lisp list $(v_0\ v_1\ v_2\ \ldots\ v_n)$ with spine $\{l_0 \ldots l_n\}$ and $v\,;\mu_0 \in \mathbb{M}_{sexp}$ then

$$\mathit{iterative{:}append}(l_0,\ v)\,;\mu_0 \gg l_0^*\,;\mu_{n+1}$$

where $\delta_{\mu_{n+1}} = \delta_{\mu_0} \cup \{l_0^*,\ \ldots, l_n^*\}$ and $l_i^* \neq l_j^* \notin \delta_{\mu_0}$ for $i \neq j$, $i,j \in n+1$. Furthermore,

1. $\mu_{n+1}(l_i^*) = [v_i,\ l_{i+1}^*]$ for $i \leq n$
2. $\mu_{n+1} = \mu_0$ on δ_{μ_0}.

Corollary 2: If $l_0\,;\mu_0$ is as above and $v\,;\mu_0$ represents the list $(w_0,\ldots,w_m)$ with spine $\{l_{n+1} \ldots l_{n+m+1}\}$ then $\mathit{iterative{:}append}(l_0,\ v)\,;\mu_0 \gg l_0^*\,;\mu_{n+1}$ and $l_0^*\,;\mu_{n+1}$ represents the list $(v_0,\ldots,v_n,w_0,\ldots,w_m)$ with spine $\{l_0^* \ldots l_n^*\ l_{n+1} \ldots l_{n+m+1}\}$.

Proof of Theorem 3: For $1 \leq i \leq n$ we define μ_i and μ_i^* by $\mu_1 = \mu_1^*$ and for $i > 0$ $l_i^*\,;\mu_i^* = \mathit{cons}(v_i,v)\,;\mu_i$ and $\mu_{i+1} = \mathit{setcdr}(l_{i-1}^*, l_i^*\,;\mu_i^*)$. We prove by induction on i that for $i \leq n$

P1. $\mathit{iterative{:}append}(l_0,v)\,;\mu_0 \gg \mathit{it{:}app}(v,l_0^*,l_i^*,l_{i+1})\,;\mu_{i+1}$

P2. $\mu_{i+1} = \mu_0$ on δ_{μ_0}

where by abuse of notation we let $l_{n+1} = \mathtt{NIL}$. Note that according to the definitions, $\delta_{\mu_{i+1}} = \delta_{\mu_i} \cup \{l_i^*\}$ with $l_i^* \notin \delta_{\mu_i}$.

Base Case $i = 0$: In this case we have by computation

$$\begin{aligned}
&\mathit{iterative{:}append}(l_0, v)\ ;\ \mu_0 \gg \\
&\quad \gg \mathtt{let}\{w \leftarrow \mathit{cons}(\mathit{car}(l_0), v)\}\mathit{it{:}app}(v, w, w, \mathit{cdr}(l_0))\ ;\ \mu_0 \\
&\quad \gg \mathit{it{:}app}(v, l_0^*, l_0^*, l_1)\ ;\ \mu_1
\end{aligned}$$

where $l_0^*\ ;\ \mu_1 = \mathit{cons}(v_0, v)\ ;\ \mu_0$.

Induction step: Suppose P1 and P2 hold for $0 \leq i' < i$ Then by computation

$$\begin{aligned}
&\mathit{it{:}app}(v, l_0^*, l_{i-1}^*, l_i)\ ;\ \mu_i \gg \\
&\quad \gg \mathit{it{:}app}(v, l_0^*, \mathit{cdr}(\mathit{rplacd}(l_{i-1}^*, \mathit{cons}(\mathit{car}(l_i), v))), \mathit{cdr}(l_i))\ ;\ \mu_i \\
&\quad \gg \mathit{it{:}app}(v, l_0^*, l_i^*, l_{i+1})\ ;\ \mathit{setcdr}(l_{i-1}^*, l_i^*\ ;\ \mu_i^*) \\
&\quad = \mathit{it{:}app}(v, l_0^*, l_i^*, l_{i+1})\ ;\ \mu_{i+1}
\end{aligned}$$

and clearly μ_{i+1} satisfies P1 and P2. $\square_{P1,P2}$

Now 2 is clearly true so it suffices to show 1. Since μ_{i+1} differs from μ_i only on l_{i-1}^* and on l_i^* we have

$$\mu_{i+1}(l_{i-1}^*) = \ldots = \mu_k(l_{i-1}^*)$$

for $i > 1$ and $k > i + 1$. Thus $\mu_{n+1}(l_i) = [v_1, l_{i+1}]$ for $i \leq n$.

$\square_{\textbf{Theorem 3}}$

5.3. Example 3: A Sophisticated Length function.

In this example we deal with a length function that not only calculates the length of a list, but also detects whether the list is *infinite*. A reference to it may be found in [C].

```
elength(l) ← elen(l,l,0)
elen(slow,fast,n) ←
    if(fast,
       if(cdr(fast),
          if(eq(fast,slow),
             if(eq(n,0),
                elen(cdr(slow),cdr(cdr(fast)),n + 2),
                INFINITY),
             elen(cdr(slow),cdr(cdr(fast)),n + 2)),
          add1(n)),
       n)
```

The key fact about *elength* is given by the following theorem.

Theorem: If $v\,;\mu \in \mathbb{M}_{elist}$ then

$$elength(v)\,;\mu = \begin{cases} length(v)\,;\mu & \text{if } v\,;\mu \in \mathbb{M}_{list} \\ \texttt{INFINITY} & \text{otherwise} \end{cases}$$

Proof of theorem: To prove that $v\,;\mu \in \mathbb{M}_{list}$ implies that $elength(v)\,;\mu = length(v)\,;\mu$ we leave as a simple exercise. We do the more difficult case. Suppose that

$$l_0\,;\mu \in \mathbb{M}_{elist} - \mathbb{M}_{list}.$$

This assumption implies that for $n \in \mathbb{N}$, $1^n \in \delta_{(l_0\,;\mu)}$ and $(l\,;\mu)_{1^n} \in \mathbb{L}$. Consequently, letting $l_i = (l_0\,;\mu)_{1^i}$ we have by the finiteness of $\delta_{(l_0\,;\mu)}$ that

$$\{[m_0,m_1] \in \mathbb{N}^{(2)} \mid m_1 > 0 \text{ and } l_{m_0} = l_{m_0+m_1}\}$$

is non-empty. Now choose $[m_0,m_1]$ to be the lexicographically least element of this set, and put x to be the smallest solution to the integer equation

$$0 = m_0 + x \qquad [\text{mod } m_1]$$

Now observe that while $l_j \neq l_{2j}$ for $0 < j < i$ we have that

$$elen(l_0,l_0,0)\,;\mu \gg elen(l_i,l_{2i},2i)\,;\mu$$

Letting $k = m_0 + x$ we claim

1. $l_k = l_{2k}$, and
2. $l_j \neq l_{2j}$ for $0 < j < k$.

It is easy to verify that, by our choice of notation, 1. is equivalent to

$$k = 2k \qquad [\text{mod } m_1]$$

which is true by virtue of our choice of x. Now suppose there was a j with $0 < j < k$ and $l_j = l_{2j}$, then by our choice of notation we would have

$$0 = j \qquad [\text{mod } m_1]$$

Now if $j < m_0$ then we would contradict our choice of $[m_0,m_1]$, on the other hand if $m_0 \leq j < m_0 + x$ then we would contradict our choice of x. Consequently no such j exists and we are done.

□Theorem

5.4. Example 4: The traditional recursive copy program.

In this example we deal with our first copying algorithm, the traditional recursive one that one learns about in introductory Lisp courses.

```
recursive:copy(u) ←
    if(atom(u),
       u,
       cons(recursive:copy(car(u)),
            recursive:copy(cdr(u))))
```

We leave the proof of the following as an exercise as it is a simple induction on $|\mathbf{Cells}^{\leq}_{\mu}(l)|$.

Theorem 4: If $l\,;\mu \in \mathbb{M}_{wfsexp}$ then

$$\mathit{recursive{:}copy}(l)\,;\mu \gg l^*\,;\mu^*$$

such that

1. $l\,;\mu \equiv l^*\,;\mu^*$
2. $\mathbf{Cells}_{\mu}(l) \cap \mathbf{Cells}_{\mu^*}(l^*) = \emptyset$
3. $|\mathbf{Cells}_{\mu}(l)| \leq |\mathbf{Cells}_{\mu^*}(l^*)|$

In general this is not the most useful copying algorithm. It has three obvious defects.

- Firstly *recursive:copy* only constructs a copy which is Lisp equal ($\equiv$) but not necessarily isomorphic ($\cong$) to the original. In fact the copy obtained by using this recursive program is the *least compact* S-expression (up to isomorphism) which is Lisp equal to the original. By least compact we mean that the copy will possess no cellular structure sharing. So, for some suitable $l\,;\mu$ we actually have that

$$|\mathbf{Cells}_{\mu^*}(l^*)| \;= 2^{|\mathbf{Cells}_{\mu}(l)|} - 1.$$

- Secondly, *recursive:copy* will not terminate on, let alone copy, cyclic S-expressions.

- Finally, its recursive nature means that it will use up stack proportional to the maximum depth of its argument, and so on large structures it may run out of free storage. Also since it does not recognize shared structure it will often duplicate calls to itself.

One of the aims of this paper is to prove the correctness of a copying algorithm that does not have these defects. We should remark that this copy algorithm does have one nice theoretical feature, namely

Proposition 5: For any $v_0\,;\mu, v_1\,;\mu \in \mathbb{M}_{wfsexp}$ we have $v_0\,;\mu \equiv v_1\,;\mu$ if and only if $\mathit{recursive{:}copy}(v_0)\,;\mu \cong \mathit{recursive{:}copy}(v_1)\,;\mu$.

6. The Correctness of the Robson Marking algorithm.

The first program that operates on $\mathbb{M}_{sexp}$ which we shall deal with is a marking algorithm. We have called it the Robson marking algorithm since it is essentially phase one of the Robson copying algorithm, [**R**]. It is interesting in its own right since it is a more sophisticated algorithm than the Deutsch-Shorr-Waite marking algorithm, [**D**], [**SW**]. Although in our domain $\mathbb{M}_{sexp}$ there are no mark or field bits, this is of no particular importance since we shall use abstract syntax [Mc] to hide this fact. The advantage of this is that we can isolate the necessary properties of the implementations of the abstract syntax that are required in the correctness proof. Thus, given a particular implementation of the algorithm we can simply check the correctness of the program by checking that the abstract syntax has the desired properties. We shall only be interested in one particular interpretation in this paper since the second phase of the Robson copying algorithm makes use of our particular implementation. An elegant treatment of the Shorr-Waite marking algorithm in a world where locations have mark bits can be found in [**T**].

The Robson marking algorithm, like the Deutsch-Shore-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. An succinct treatment of pointer reversal or pointer rotation, as it is sometimes called, may be found in [S], although the notation in that paper has an unfortunate tendency to confuse control and data.

6.1. The Robson Marking Algorithm

In the Robson marking algorithm the process of marking a cell consists of allocating a new cell and moving into this new cell the contents of the cell being marked. The cell being marked is then updated so that its car contains a mark and its cdr points to the new cell. A mark is an object specially allocated before marking and so recognizably not part of the structure to be marked. We use seven different marks to store more information than just simply whether or not the cell has been seen before. We shall denote these marks by `ER,EL,E10,M00,M01,M10,M11`. Their meaning roughly being described by

`EL` - Exploring the left hand side of the cell. If the car is not terminal, then while it is being marked the pointer to it will be utilized to store the previous stack, the cell itself then becomes the current stack.

`E10` - The left hand side is atomic or has been visited before, now exploring the right hand side.

`ER` - Exploring the right hand side after having explored the left hand side, which was neither atomic nor already marked. If the cdr is not terminal, then while it is being marked the pointer to it will be utilized to store the previous stack, the cell itself then becomes the current stack.

M11 - Both the left and right hand side are either atoms or cells that were visited earlier in the left first scan, such cells are called terminal.

M01 - Only the right hand side was terminal.

M10 - Only the left hand side was terminal.

M00 - Neither the left nor the right were terminal, and both sides have been completely investigated.

In addition there is a mark ALPHA that indicates the bottom of the stack, initially also the top. A cell that is marked either EL, ER or E10 resides on the stack, the inverted pointer chain. Marks may be either atoms or cells. The crucial point is that they must be distinct from one another and disjoint from the structure being marked. This will be assumed in the following.

The actual definitions of the Robson algorithm are:

```
rmark(s) ← if(atom(s),s,markcar(s,ALPHA))
markcar(s,stack) ←
    [mkmark(s,EL),
     let{t1 ← a(s)}
        if(terminal(t1),
           [setm(s,E10),markcdr(s,stack)],
           [seta(s,stack),markcar(t1,s)])]
markcdr(s,stack) ←
    let{t2 ← d(s)}
       if(terminal(t2),
          ifs(eq(ER,m(s)),[setm(s,M01),popstack(s,stack)],
              eq(E10,m(s)),[setm(s,M11),popstack(s,stack)])
          [setd(s,stack),markcar(t2,s)])
popstack(s,stack) ←
    if(eq(stack,ALPHA),
       s,
       let{t1 ← a(stack),t2 ← d(stack)},
          ifs(eq(EL,m(stack)),
              [setm(stack,ER),seta(stack,s),markcdr(stack,t1)],
              eq(ER,m(stack)),
              [setm(stack,M00),setd(stack,s),popstack(stack,t2)],
              eq(E10,m(stack)),
              [setm(stack,M10),setd(stack,s),popstack(stack,t2)]))
```

The program as written above is a tail recursive definition, which uses the abstract syntax

$$m, a, d, seta, setd, mkmark, setm, marked, terminal$$

The function *mkmark* does the job of allocating the new cell and placing the contents of the original cell in it, altering the original so that its car contains the appropriate mark and its cdr the new cell. *a* and *d* then access the old car and cdr, while *seta* and *setd* update them. *setm* just replaces the mark without allocating any new cells. *marked* determines whether the cell is marked and *m* returns the mark. *terminal* just checks whether a cell is terminal, namely whether it is an atom or an already marked cell.

To be explicit we have the following definitions of these functions.

$m(\mathtt{l}) \leftarrow car(\mathtt{l})$

$a(\mathtt{l}) \leftarrow car(cdr(\mathtt{l}))$

$d(\mathtt{l}) \leftarrow cdr(cdr(\mathtt{l}))$

$mkmark(\mathtt{l},\mathtt{m}) \leftarrow \mathtt{let}\{\mathtt{t2} \leftarrow car(\mathtt{l})\}[rplaca(\mathtt{l},\mathtt{m}), rplacd(\mathtt{l}, cons(\mathtt{t2}, cdr(\mathtt{l})))]$

$setm(\mathtt{l},\mathtt{m}) \leftarrow rplaca(\mathtt{l},\mathtt{m})$

$seta(\mathtt{l},\mathtt{x}) \leftarrow rplaca(cdr(\mathtt{l}),\mathtt{x})$

$setd(\mathtt{l},\mathtt{x}) \leftarrow rplacd(cdr(\mathtt{l}),\mathtt{x})$

$marked(\mathtt{l}) \leftarrow memq(car(\mathtt{l}),(\mathtt{ER},\mathtt{EL},\mathtt{E10},\mathtt{M11},\mathtt{M01},\mathtt{M10},\mathtt{M00}))$

$terminal(\mathtt{l}) \leftarrow \mathtt{or}(atom(\mathtt{l}), marked(\mathtt{l}))$

The final product of this program will be specified in more detail later, for now the following is sufficient. After *rmark*-ing an S-expression, all cells accessible from the S-expression have been destructively altered so that their car contains a mark and their cdr points to a new cell that contains the original contents. In other words if $rmark(l);\mu \gg v;\mu^*$ then for $l_i \in \mathbf{Cells}_\mu(l)$ we have

$$car(l_i)\,;\mu \gg v_a\,;\mu \wedge a(l_i)\,;\mu^* \gg v_a\,;\mu^* \quad \text{and} \quad cdr(l_i)\,;\mu \gg v_d\,;\mu \wedge d(l_i)\,;\mu^* \gg v_d\,;\mu^*.$$

Definition of $(l\,;\mu)_a$, $(l\,;\mu)_d$: We write $(l\,;\mu)_a$ to denote the value of $a(l)\,;\mu$, $(l\,;\mu)_d$ that of $d(l)\,;\mu$, and $(l\,;\mu)_m$ that of $m(l)\,;\mu$. Thus, given the above interpretation of the abstract syntax, when a, d or m appears as the argument to $(l\,;\mu)$ it can simply be taken to denote 10, 11 or 0 respectively. Often when μ is fixed by some context we simply write v_a and v_d leaving μ as understood.

Aside: As an aside we describe a memory structure over which we can model the usual low level implementation of a marking algorithm, such as is used in mark and sweep garbage

collection. In this version we work over a memory structure that one obtains by adding a mark bit to a cell, $\mathbb{M}_{msexp}$, in short.

$$\mathbb{M}_{msexp} = \{\mu \in \mathbb{M}_{(\mathbb{A},\mathbb{L})} \mid \mu \in [\delta_\mu \rightarrow \mathbb{V}^{(3)}]\}$$
$$\mathbb{O}_{msexp} = \mathbb{O}_{sexp} \cup \{m, setm, mkmark\}$$

Over this structure *car* and *cdr* access the second and third elements and *m* returns the first. *cons* returns a new label with the mark bit set initially to a default value NIL. *setm* and *mkmark* simply update the first bit and *rplacx* updates the *cxr* part for $x \in \{a, d\}$. In this version of the program the result of marking $v\,;\mu$ leaves the *car-cdr* structure of $v\,;\mu$ unchanged, the only modification is that the mark bits in the structure now contain the appropriate information concerning the left-first spanning tree. Using the above notation we have that in this version

$$\mu(l) = [(l\,;\mu)_m,\ (l\,;\mu)_a,\ (l\,;\mu)_d].$$

We now return to the subject at hand. Here and elsewhere we shall make a habit of ignoring the value returned by *mkmark*, *seta*, and *setd*, treating them as being analogous to *setcar* and *setcdr*. This should not cause confusion since none of the programs in this paper will ever make use of the value returned by such an operation. The following are the properties of the abstract syntax that are required in the proof. For expositiory purposes we give them names.

Cancellation: For $x \in \{a, d\}$ and $l\,;\mu$ marked we have

$$setx(l, v_2\,; setx(l, v_1\,; \mu)) = setx(l, v_2\,; \mu)$$

and

$$setm(l, v_2\,; mkmark(l, v_1\,; \mu)) = mkmark(l, v_2\,; \mu)$$

Absorption: If $x \in \{a, d, m\}$ and $v = (l\,;\mu)_x$ then $setx(l, v\,;\mu) = \mu$.

Commutativity: If $x, y \in \{a, d, m\}$ with $x \neq y$ when $l = l^*$, l, l^* are both marked, and neither $(l\,;\mu)_1 = l^*$ nor $(l^*\,;\mu)_1 = l$ then

$$setx(l, v\,; sety(l^*, v^*\,; \mu)) = sety(l^*, v^*\,; setx(l, v\,; \mu)).$$

Access: Finally for $x \in \{a, d, m\}$ and l marked we have

$$x(l)\,; setx(l, v\,; \mu) \gg v\,; setx(l, v\,; \mu)$$

and when l is not marked

$$m(l)\,; mkmark(l, v\,; \mu) \gg v\,; mkmark(l, v\,; \mu)$$

6.2. Methods of recursion and proof

We now commence with the preliminaries that are required to prove the correctness of the Robson marking and copying algorithms. We first define the notion of a spanning tree of a graph. The idea here being that to define a function recursively on a graph one must choose a path and an order in which to visit the cells, and to prevent looping one must have some means of knowing when to stop. The first problem gives rise to spanning trees whilst the second is handled, as we have already mentioned, by modifying the cells as we visit them so that we can recognize an *already visited* cell when we see one. Of course these two problems are not unrelated.

In the correctness proofs, of both the copying and marking algorithms, the essential idea is the same. We define a memory transformation recursively, which under certain natural pre-conditions corresponds to the result of a simple recursively defined function in our computation theory. These simple programs are quite inefficient in the sense that they are not tail recursive and use up stack proportional to the size of their argument. They do however have the advantage that they are very easy to to understand. We then prove, again under natural pre-conditions, that the simple recursive program computes the same partial function as its *pointer reversing* counterpart. This is done by using the transformation mentioned above. These *pointer reversing* programs consist of a set of mutually tail recursive functions and thus use no stack. We should emphasize that we have included the simple recursive versions purely for motivation. They are in no way *logically necessary* for the actual proofs.

All proofs, not surprisingly, are by induction. Consequently we must find some measure which gets smaller as the program progresses. It is here that the **TC** construction, of section 4, comes in handy. In both cases we can use a variant of **TC** to define a type of subset, of $\mathbf{Cells}^{<}$, that measures the progress of the algorithms.

6.3. Spanning trees

For $l\,;\mu \in \mathbb{M}_{sexp}$ we say that X is a *connected* subset of $\mathbf{Cells}_{\mu}(l)$ if X is the image of a subtree of $\mathbb{T}$ under the map $(l\,;\mu)$. So, for example, subsets defined by the **TC** operation are connected. For X, a connected subset of $\mathbf{Cells}_{\mu}(l)$, we define a *spanning tree* for X at $l\,;\mu$ to be a set $S \subset \mathbb{T}$ having the following properties

1. $(\forall v \in X)(\exists!\, \sigma \in S)(l\,;\mu)_{\sigma} = v$, and
2. S is a subtree of $\mathbb{T}$.

For convenience we say that a cell l_i is left (right) *terminal* with respect to a spanning tree S (at $l;\mu$) if $\exists \sigma \in S \;\; l_i = (l;\mu)_{\sigma}$ but $\sigma \diamond 0$ ($\sigma \diamond 1$) is not an element of S. For example in the Robson marking algorithm we use terminal to mean terminal with respect to the left-first spanning tree. There are various well known spanning trees for graphs, [**A**]. We shall be using the left-first spanning tree in this paper. The left-first spanning tree of $\mathbf{Cells}_{\mu}(v)$ can

be defined as follows. For $l \in \mathbf{Cells}_\mu(v)$ the function $\mathbf{Left}_{v;\mu}$:$[\mathbf{Cells}_\mu(v) \rightarrowtail \mathbb{T}]$ chooses the least path in μ from v to l with respect to the Brouwer-Kleene ordering ($\preceq$).

$$\mathbf{Left}_{v;\mu}(l) = \sigma \;\rightarrow\; (v\,;\mu)_\sigma = l \wedge \forall\sigma_0((v\,;\mu)_{\sigma_0} = l \rightarrow \sigma \preceq \sigma_0).$$

The left first spanning tree of $v\,;\mu$ is then the image of $\mathbf{Left}_{v;\mu}$ and is denoted by $\Lambda_{v;\mu}$.

$$\Lambda_{v;\mu} \underset{\text{df}}{=} \{\mathbf{Left}_{v;\mu}(l) \mid l \in \mathbf{Cells}_\mu(v)\}$$

Now given that S is a spanning tree for X at $l\,;\mu$ and $l_0 \in X$, we say that l_1 *lies below* l_0 in S if $\exists\sigma_0,\sigma_1 \in \mathbb{T}$ such that

1. $\sigma_0,\sigma_1 \in S$
2. $(l\,;\mu)_{\sigma_i} = l_i$, for $i \in 2$, and
3. $\sigma_0 \leq \sigma_1$ in $\mathbb{T}$.

Similarly we can talk about l_0 being above, to the left, or to the right of l_1 in S. We also put

$$S(l_0) = \{l_1 \mid l_1 \text{ lies below } l_0 \text{ in } S\}.$$

Observe that $S(l_0) \subseteq X$ and that if l_1 lies below l_0 in S then $S(l_1) \subseteq S(l_0)$ with equality holding only when $l_0 = l_1$.

6.4. The recursive Robson marking program and transformation

We now define the simple recursive program *rec:rmark*, a straight forward left-first recursive marking algorithm. Thus *rec:mark* traverses the graph by following the left-first spanning tree, Λ, in the Brouwer-Kleene ordering.

```
rec:rmark(s) ← if(atom(s),s,[rec:rmark1(s),s])
rec:rmark1(s) ←
    [mkmark(s,EL),
     if(terminal(a(s)),
        if( terminal(d(s)),
            setm(s,M11),
            [setm(s,M10),rec:rmark1(d(s))])
        [rec:rmark1(a(s)),
         if(terminal(d(s)),
            setm(s,M10),
            [setm(s,M00),rec:rmark1(d(s))])])]
```

The set that we use induction on to prove properties of *rec:mark* and of its *pointer reversal* counterpart is $\mathbf{Unmarked}_\mu(l)$. It consists of all those cells that are unmarked and are reachable from l via paths through unmarked cells.To be precise:

Definition of $\mathbf{Unmarked}_\mu(l)$:

$$\mathbf{Unmarked}_\mu(l) \underset{\text{df}}{=} \mathbf{TC}(l\,;\mu, \neg\Phi_T)$$

where $\Phi_T(v\,;\mu)$ iff $v \in \mathbb{A}$ or $\Phi_M(v\,;\mu)$, and $\Phi_M(v\,;\mu)$ iff $\mathit{marked}(v)\,;\mu \gg \mathtt{T}\,;\mu$.

The transformation on memory objects that we use to prove that ***rmark*** and ***rec:rmark*** agree can now be defined. We first state some assumptions that are needed to ensure that the transformation is well-defined. Since we will often make use of these assumptions we give them a name, **RM** condition.

RM condition: Λ is the left first spanning tree for $\mathbf{Unmarked}_\mu(l)$ at $l;\mu$, $l^*;\mu^*$ is such that $l^* \in \mathbf{Unmarked}_\mu(l)$, and $\mu^* = \mu$ on all cells, including l^*, that lie below l^* in Λ.

Definition of RM: We define the transformation $\mathbf{RM}^\Lambda_{\mu^*}(l^*)$ on memories recursively as follows:

$$\mathbf{RM}^\Lambda_{\mu^*}(l^*) = \begin{cases} \mathit{mkmark}(l^*,\mathtt{M11}\,;\mu^*) & \text{if } l^* \text{ is both left and right terminal w.r.t } \Lambda \\ \mathbf{RM}^\Lambda_{\mathit{mkmark}(l^*,\mathtt{M01};\mu^*)}(l^*_a) & \text{if } l^* \text{ is right but not left terminal w.r.t } \Lambda. \\ \mathbf{RM}^\Lambda_{\mathit{mkmark}(l^*,\mathtt{M10};\mu^*)}(l^*_d) & \text{if } l^* \text{ is left but not right terminal w.r.t } \Lambda. \\ \mathbf{RM}^\Lambda_{\mathbf{RM}^\Lambda_{\mu^{**}}(l^*_a)}(l^*_d) & \text{if } l^* \text{ is neither left nor right terminal w.r.t } \Lambda. \end{cases}$$

where $\mu^{**} = \mathit{mkmark}(l^*,\mathtt{M00}\,;\mu^*)$ and $l^*_x = (l^*\,;\mu^*)_x$ for $x \in \{a,d\}$. We have the following simple properties of $\mathbf{RM}_{\mu^*}(l^*)$

Proposition 5: If Λ is the left first spanning tree for $\mathbf{Unmarked}_\mu(l)$ and $l^*;\mu^*$ satisfies the **RM** condition, then

1. $\mathbf{RM}_{\mu^*}(l^*)$ agrees with μ^* on all locations not in $\mathbf{Unmarked}_\mu(l^*)$
2. $(l_i\,;\mathbf{RM}_{\mu^*}(l^*))_a = (l_i\,;\mu^*)_0$ if $l_i \in \mathbf{Unmarked}_\mu(l^*)$

   $(l_i\,;\mathbf{RM}_{\mu^*}(l^*))_d = (l_i\,;\mu^*)_1$ if $l_i \in \mathbf{Unmarked}_\mu(l^*)$
3. If l_i lies below l^* in Λ then

$$(l_i\,;\mathbf{RM}_{\mu^*}(l^*))_m = \begin{cases} \mathtt{M00} & l_i \text{ is neither left nor right terminal w.r.t } \Lambda \\ \mathtt{M01} & l_i \text{ is right but not left terminal w.r.t } \Lambda \\ \mathtt{M10} & l_i \text{ is left but not right terminal w.r.t } \Lambda \\ \mathtt{M11} & l_i \text{ is both left and right terminal w.r.t } \Lambda \end{cases}$$

Proof of proposition 5: This is by induction on $|\Lambda(l^*)|$.

Base case: $|\Lambda(l^*)| = 1$, in this case we know that l^* is both left and right terminal in Λ. Consequently $\mathbf{RM}_{\mu^*}(l^*) = \mathit{mkmark}(l^*,\mathtt{M11}\,;\mu^*)$ and 1, 2 and 3 clearly hold. $\square_{\text{base case}}$

Induction step: $|\Lambda(l^*)| > 1$. There are three cases to consider, we shall only do the case when l^* is neither left nor right terminal; the other two cases being somewhat simpler

versions of the same argument. So assuming that l^* is neither left nor right terminal with respect to Λ we have

$$\mathbf{RM}_{\mu^*}(l^*) = \mathbf{RM}_{\mathbf{RM}_{\mu^{**}}(l_a^*)}(l_d^*)$$

where $\mu^{**} = \mathit{mkmark}(l^*, \mathtt{M00}\,;\mu^*)$ and $l_x^* = (l^*\,;\mu^*)_x$ for $x \in \{a,d\}$.

Now $l_a^*\,;\mu^{**}$ satisfies $|\Lambda(l_a^*)| < |\Lambda(l^*)|$ and using the fact that $\mu = \mu^*$ on $\Lambda(l^*)$ we have putting

$$\mu_a = \mathbf{RM}_{\mu^{**}}(l_a^*)$$

that μ_a satisfies 1, 2, and 3 on $\mathbf{Unmarked}_{\mu}(l_a^*)$, by the induction hypothesis. And again since $|\Lambda(l_d^*)| < |\Lambda(l^*)|$ and the fact that $\Lambda(l_a^*) \cap \Lambda(l_d^*) = \emptyset$ we have that $\mu_a = \mu$ on $\Lambda(l_d^*)$ so the induction hypothesis allows us to conclude that

$$\mu_d = \mathbf{RM}_{\mu_a}(l_d^*)$$

satisfies 1, 2, and 3. A simple argument puts these together to show that μ_d satisfies 1, 2, and 3 on $\{l^*\} \oplus \Lambda(l_a^*) \oplus \Lambda(l_d^*)$

□proposition 5

A further useful fact is the following, the proof of which is a simple induction on $|\Lambda(l^*)|$.

Commutativity lemma for $\mathbf{RM}_{\mu^*}(l^*)$: If $l^*\,;\mu^*$ satisfies the hypothesis of the definition of $\mathbf{RM}_{\mu^*}(l^*)$ and Γ is a memory operation of the form $\lambda\mu.\,\mathit{setx}(l_k, v\,;\mu)$ where $x \in \{m,a,d\}$ and $l_k \notin \mathbf{Unmarked}_{\mu^*}(l^*)$ then

$$\Gamma(\mathbf{RM}_{\mu^*}(l^*)) = \mathbf{RM}_{\Gamma(\mu^*)}(l^*)$$

The fact that this transformation **RM** is indeed what is computed by *rec:rmark* is verified by the following theorem, the proof of which we leave as an exercise since it is much simpler than the one that follows it.

Theorem 5: If $l\,;\mu \in \mathbb{M}_{\mathit{secp}}$ and Λ is the left first spanning tree for $\mathbf{Unmarked}_{\mu}(l)$ at $l\,;\mu$ then

$$\mathit{rec{:}rmark}(l)\,;\mu \gg l\,;\mathbf{RM}_{\mu}(l)$$

6.5. The Main marking theorem

Using the concepts defined above we can formulate the main theorem of this section as follows.

Theorem 6: If $l\,;\mu \in \mathbb{M}_{\mathit{secp}}$ and Λ is the left first spanning tree for $\mathbf{Unmarked}_{\mu}(l)$ at $l\,;\mu$ then

$$\mathit{rmark}(l)\,;\mu \gg l\,;\mathbf{RM}_{\mu}(l)$$

Theorem 6 follows from the following lemma.

Main Lemma: If $l_0 ; \mu_0 \in \mathbb{M}_{sexp}$ is such that

1. $l_0 \in \mathbf{Unmarked}_{\mu}(l)$
2. $\mu_0 = \mu$ on $\mathbf{Unmarked}_{\mu_0}(l_0)$
3. No cell below l_0 in the spanning tree Λ is marked in μ_0
4. All cells above and to the left of l_0 in Λ are marked in μ_0.

then

$$markcar(l_0, v) ; \mu_0 \gg popstack(l_0, v) ; \mathbf{RM}_{\mu_0}(l_0)$$

Proof of the main lemma: This is by induction on $|\mathbf{Unmarked}_{\mu_0}(l_0)|$.

Base case: $|\mathbf{Unmarked}_{\mu_0}(l_0)| = 1$. In this case both $(l_0 ; \mu_0)_0$ and $(l_0 ; \mu_0)_1$ are either marked or atomic, by conditions 2. and 3. this means that l_0 is both left and right terminal w.r.t Λ. Now

$$markcar(l_0, v) ; \mu_0 \gg markcdr(l_0, v) ; \mu_1$$

where $\mu_1 = setm(l_0, \mathtt{E10} ; mkmark(l_0, \mathtt{EL} ; \mu_0)) = mkmark(l_0, \mathtt{E10} ; \mu_0)$ by cancellation, furthermore

$$markcdr(l_0, v) ; \mu_1 \gg popstack(l_0, v) ; \mu_2$$

where $\mu_2 = setm(l_0, \mathtt{M11} ; \mu_1) = mkmark(l_0, \mathtt{M11} ; \mu_0) = \mathbf{RM}_{\mu_0}(l_0)$, again by cancellation. $\square_{\text{Base case}}$

Induction step: Suppose that the lemma is true for memory objects of less rank than $|\mathbf{Unmarked}_{\mu_0}(l_0)| > 1$. We split this part of the proof into three cases. For convenience we will let v_a and v_d be $(l_0 ; \mu_0)_0$ and $(l_0 ; \mu_0)_1$ respectively.

Case 1. $\Phi_T(v_a ; \mu_0) \wedge \neg\Phi_T(v_d ; \mu_0)$

Case 2. $\neg\Phi_T(v_a ; \mu_0) \wedge \Phi_T(v_d ; \mu_0)$

Case 3. $\neg\Phi_T(v_a ; \mu_0) \wedge \neg\Phi_T(v_d ; \mu_0)$

Case 1: In this case we know that l_0 is left terminal. We also know that $v_d \in \mathbb{L}$. So

$$markcar(l_0, v) ; \mu_0 \gg markcdr(l_0, v) ; \mu_1$$

where $\mu_1 = mkmark(l_0, \mathtt{E10}; \mu_0)$ now if $v_d = l_0$ then l_0 is in fact both left and right terminal and in this case

$$markcdr(l_0, v) ; \mu_1 \gg popstack(l_0, v) ; \mu_2$$

where $\mu_2 = setm(l_0, \mathtt{M11}; \mu_1) = mkmark(l_0, \mathtt{M11}; \mu_0) = \mathbf{RM}_{\mu_0}(l_0)$, by cancellation. Suppose that $v_d \neq l_0$, which by the conditions of the lemma means that l_0 is left but not right terminal w.r.t Λ, then

$$markcdr(l_0, v) ; \mu_1 \gg markcar(v_d, l_0) ; \mu_2$$

where $\mu_2 = setd(l_0, v\ ;\ \mu_1)$. It is a simple task to show that $v_d\ ;\ \mu_2$ satisfies the conditions of the lemma and that $|\mathbf{Unmarked}_{\mu_2}(v_d)| < |\mathbf{Unmarked}_{\mu_0}(l_0)|$. By induction

$$markcar(v_d, l_0)\ ;\ \mu_2 \gg popstack(v_d, l_0)\ ;\ \mu_3$$

where $\mu_3 = \mathbf{RM}_{\mu_2}(v_d)$. As $l_0 \notin \mathbf{Unmarked}_{\mu_2}(v_d)$ we know that l_0 is not altered in the transition from μ_2 to μ_3. Thus

$$popstack(v_d, l_0)\ ;\ \mu_3 \gg popstack(l_0, v)\ ;\ \mu_4$$

where $\mu_4 = setd(l_0, v_d\ ;\ setm(l_0, \texttt{M10}\ ;\ \mu_3))$ By the commutativity lemma we have

$$setd(l_0, v_d\ ;\ setm(l_0, \texttt{M10}\ ;\ \mathbf{RM}_{\mu_2}(v_d))) = \mathbf{RM}_{setd(l_0, v_d; setm(l_0, \texttt{M10}; \mu_2))}(v_d)$$

where

$$setd(l_0, v_d\ ;\ setm(l_0, \texttt{M10}\ ;\ \mu_2)) = setd(l_0, v_d\ ;\ setm(l_0, \texttt{M10}\ ;\ setd(l_0, v\ ;\ mkmark(l_0, \texttt{E10}\ ;\ \mu_0))))$$

but by cancellation and absorption this is just $mkmark(l_0, \texttt{M10}; \mu_0)$ and thus $\mu_4 = \mathbf{RM}_{\mu_0}(l_0)$. $\square_{\textbf{case 1}}$

Case 2: In this case we know that l_0 is right terminal w.r.t Λ. We have two possibilities either $v_a = l_0 \vee v_a \neq l_0$. If $v_a = l_0$ then l_0 is both left and right terminal and

$$markcar(l_0, v)\ ;\ \mu_0 \gg markcdr(l_0, v)\ ;\ \mu_1$$

where $\mu_1 = mkmark(l_0, \texttt{E10}\ ;\ \mu_0)$. Since v_d is either marked or atomic

$$markcdr(l_0, v)\ ;\ \mu_1 \gg popstack(l_0, v)\ ;\ \mu_2$$

where $\mu_2 = setm(l_0, \texttt{M11}\ ;\ \mu_1) = mkmark(l_0, \texttt{M11}\ ;\ \mu_0) = \mathbf{RM}_{\mu_0}(l_0)$.

If $v_a \neq l_0$ then

$$markcar(l_0, v)\ ;\ \mu_0 \gg markcar(v_a, l_0)\ ;\ \mu_1$$

where $\mu_1 = seta(l_0, v\ ;\ mkmark(l_0, \texttt{EL}\ ;\ \mu_0))$. Now by the induction hypothesis

$$markcar(v_a, l_0)\ ;\ \mu_1 \gg popstack(v_a, l_0)\ ;\ \mu_2$$

where $\mu_2 = \mathbf{RM}_{\mu_1}(v_a)$. As $l_0 \notin \mathbf{Unmarked}_{\mu_1}(v_d)$, its contents remains unchanged during this transition, consequently

$$popstack(v_a, l_0)\ ;\ \mu_2 \gg markcdr(l_0, v)\ ;\ \mu_3$$

where $\mu_3 = seta(l_0, v_a\ ;\ setm(l_0, \texttt{ER}\ ;\ \mu_2))$. Finally since v_d is atomic or marked in μ_0 and consequently remains so in μ_3 we have

$$markcdr(l_0, v)\ ;\ \mu_3 \gg popstack(l_0, v)\ ;\ \mu_4$$

where $\mu_4 = \mathit{setm}(l_0,\texttt{MO1} ; \mu_3)$. Now

$$\mu_4 = \mathit{setm}(l_0,\texttt{MO1} ; \mathit{seta}(l_0,v_a ; \mathit{setm}(l_0,\texttt{ER} ; \mathbf{RM}_{\mu_1}(v_a)))).$$

By the commutativity lemma we have $\mu_4 = \mathbf{RM}_{\mu^*}(v_a)$ where

$$\mu^* = \mathit{setm}(l_0,\texttt{MO1} ; \mathit{seta}(l_0,v_a ; \mathit{setm}(l_0,\texttt{ER} ; \mathit{seta}(l_0,v ; \mathit{mkmark}(l_0,\texttt{EL} ; \mu_0)))))$$

which by cancellation, commutativity and absorption reduces to $\mathit{mkmark}(l_0,\texttt{MO1} ; \mu_0)$ and so $\mu_4 = \mathbf{RM}_{\mu_0}(l_0)$. $\square_{\textbf{case 2}}$

Case 3: In this case neither v_a nor v_d is atomic or marked, however there are several possibilities (i) $l_0 \neq v_a \neq v_d \neq l_0$, (ii) $l_0 = v_a = v_d$ (iii) $l_0 \neq v_a = v_d$ (iv) $l_0 = v_a \neq v_d$, and (v) $l_0 = v_d \neq v_a$.

The last four cases all reduce to ones already considered so we leave them to the suspicious reader to verify. In the first case we have

$$\mathit{markcar}(l_0,v) ; \mu_0 \gg \mathit{markcar}(v_a,l_0) ; \mu_1$$

where $\mu_1 = \mathit{seta}(l_0,v ; \mathit{mkmark}(l_0,\texttt{EL} ; \mu_0))$. Now by the induction hypothesis we have

$$\mathit{markcar}(v_a,l_0) ; \mu_1 \gg \mathit{popstack}(v_a,l_0) ; \mu_2$$

where $\mu_2 = \mathbf{RM}_{\mu_1}(v_a)$. And as $l_0 \notin \mathbf{Unmarked}_{\mu_1}(v_d)$

$$\mathit{popstack}(v_a,l_0) ; \mu_2 \gg \mathit{markcdr}(l_0,v) ; \mu_3$$

where $\mu_3 = \mathit{seta}(l_0,v_a ; \mathit{setm}(l_0,\texttt{ER} ; \mu_2))$. Now, if v_d is marked in μ_3 then it must occur below v_a in the spanning tree, in which case l_0 is right but not left terminal. Given that v_d is marked in μ_3 we have that

$$\mathit{markcdr}(l_0,v) ; \mu_3 \gg \mathit{popstack}(l_0,v) ; \mu_4$$

where $\mu_4 = \mathit{setm}(l_0,\texttt{MO1} ; \mu_3)$ and, just as in case 2, $\mu_4 = \mathbf{RM}_{\mu_0}(l_0)$. So suppose that v_d is not marked in μ_3. In other words suppose that l_0 is neither left nor right terminal. Then

$$\mathit{markcdr}(l_0,v) ; \mu_3 \gg \mathit{markcar}(v_d,l_0) ; \mu_4$$

where $\mu_4 = \mathit{setd}(l_0,v;\mu_3)$. Consequently using the induction hypothesis again we have that

$$\mathit{markcar}(v_d,l_0) ; \mu_4 \gg \mathit{popstack}(v_d,l_0) ; \mu_5$$

where $\mu_5 = \mathbf{RM}_{\mu_4}(v_d)$. As l_0 is not altered in the transition from μ_4 to μ_5 we have

$$\mathit{popstack}(v_d,l_0) ; \mu_5 \gg \mathit{popstack}(l_0,v) ; \mu_6$$

where $\mu_6 = setd(l_0, v_d\ ;\ setm(l_0, \texttt{MOO}\ ;\ \mu_5))$. Now

$$\mu_4 = setd(l_0, v\ ;\ seta(l_0, v_a\ ;\ setm(l_0, \texttt{ER}\ ;\ \mathbf{RM}_{\mu_1}(v_a))))$$

and

$$\mu_6 = setd(l_0, v_d\ ;\ setm(l_0, \texttt{MOO}\ ;\ \mathbf{RM}_{\mu_4}(v_d)))$$

so by the commutativity lemma

$$\mu_6 = \mathbf{RM}_{setd(l_0, v_d; setm(l_0, \texttt{MOO}; \mu_4))}(v_d) = \mathbf{RM}_{\mu^+}(v_d)$$

where

$$\mu^+ = setd(l_0, v_d\ ;\ setm(l_0, \texttt{MOO}\ ;\ setd(l_0, v\ ;\ seta(l_0, v_a\ ;\ setm(l_0, \texttt{ER}\ ;\ \mathbf{RM}_{\mu_1}(v_a)))))).$$

Using the commutativity lemma again this becomes

$$\mu^+ = \mathbf{RM}_{setd(l_0, v_d; setm(l_0, \texttt{MOO}; setd(l_0, v; seta(l_0, v_a; setm(l_0, \texttt{ER}; seta(l_0, v; mkmark(l_0, \texttt{EL}; \mu_0))))))))}(v_a)$$

which by cancellation, commutativity and absorption is just $\mathbf{RM}_{mkmark(l_0, \texttt{MOO}; \mu_0)}(v_a)$ Thus $\mu_6 = \mathbf{RM}_{\mu_0}(l_0)$. $\square_{\texttt{case 3}}$

$\square_{\texttt{main lemma}}$

7. The Correctness of the Robson copy algorithm.

In this section we prove the correctness of the Robson copying algorithm. The section is structurally similar to the previous one. In 7.1 we define a simple recursive program, which describes the same function as the Robson copying algorithm, its *pointer reversal* counterpart. In 7.2 we introduce some notation and define the set upon which we shall perform induction. Then in 7.3 we introduce the transformation by which we prove the equivalence of our two copying programs. In 7.4 we introduce the actual Robson algorithm and finally in 7.5 we prove its correctness.

7.1. The Recursive version of the copy algorithm.

The following program is a recursive version of the Robson algorithm and we shall study it in this section as a preliminary to the actual Robson copy algorithm. It simply implements the transformation **Peel**, which will be defined shortly. We begin by a discussing how the program works. As Robson himself says of his own algorithm:

> *A new algorithm is presented which copies cyclic list structures using bounded workspace and linear time ..... The distinctive feature of this algorithm is a technique for traversing the structure twice, using the same spanning tree in each case, first from left to right and then from right to left.*

Our simple recursive version uses much the same technique, the only difference is that since our version is not tail recursive we cannot claim to use only bounded workspace. With respect to the traversals at least we have made our job somewhat easier. The first traversal of the structure corresponds precisely to the algorithm that we have called the Robson marking algorithm. Consequently we need now only describe the second traversal, best described as a *peeling* operation.

Recall that after the first traversal each cell is allocated a new cell, which we shall call its *image*. The original cell is modified so that its *car* part contains a mark denoting its place in the left-first spanning tree, while its *cdr* part contains its image. The image in turn contains the cells original contents. Consequently each original cell now contains two more pieces of information, namely whether its *car* or *cdr* is terminal in the Brouwer-Kleene ordering of the left-first spanning tree. This information allows the second traversal to use the same spanning tree, in the reverse order, without further marking. The crucial observation is that since the decision to follow a pointer depends on the mark in the cell containing it, rather than upon the cell pointed to, this traversal can remove the marks as it uses them. Furthermore since the *image* cell which is used together with the original cell to store the mark and the original contents, is no longer required, this cell can be recycled and used as the corresponding cell in the copy. This storage optimization is similar in spirit to that done recently in the study of *tail recursion up to a cons*, see for example [**W**].

```
rec:copy(l) ←
    if(atom(l),
       l,
       [rmark(l),let{t1 ← cdr(l)}[rec:peel(l),t1]])
rec:peel(oldcel) ←
    let{newcel ← cdr(oldcel)}
    let{ newcar ← image(car(newcel)),
         newcdr ← image(cdr(newcel)),
         oldcar ← car(newcel),
         oldcdr ← cdr(newcel)}
    [ifs(eq(M00,m(oldcel)),[rec:peel(oldcdr),rec:peel(oldcar)],
         eq(M01,m(oldcel)),rec:peel(oldcar),
         eq(M10,m(oldcel)),rec:peel(oldcdr),
         eq(M11,m(oldcel)),NIL),
    rplaca(oldcel,oldcar),
    rplacd(oldcel,oldcdr),
    rplaca(newcel,newcar),
    rplacd(newcel,newcdr)]
image(l) ← if(atom(l),l,cdr(l))
```

We now set about developing some concepts that enable us to prove the correctness of this as well as of the Robson algorithm.

7.2. Some additional concepts and notation.

In this section we abide by the following important notational assumptions. They correspond to the following scenario: we begin with a memory object $l_0\,;\mu_0$ with $\mathbf{Cells}_{\mu_0}(l_0) = \{l_0,\ldots,l_r\}$ such that none of these cells are marked. Thus

$$\mathbf{Unmarked}_{\mu_0}(l_0) = \mathbf{Cells}_{\mu_0}(l_0).$$

For convenience we let the *car* of $l_i\,;\mu_0$ be v_{ia} and the *cdr* be v_{id}, in other words $(l_i\,;\mu_0)_0 = v_{ia}$ and $(l_i\,;\mu_0)_1 = v_{id}$. We then apply the Robson marking algorithm to $l_0\,;\mu_0$ and so obtain $l_0\,;\mu$ where

$$rmark(l_0)\,;\mu_0 \gg l_0\,;\mu = l_0\,;\mathbf{RM}^{\Lambda}_{\mu_0}(l_0).$$

Each cell l_i for $i \in r+1$ is allocated a new cell, which we call its *image*. We denote the image of the cell $l_i \in \mathbf{Cells}_{\mu_0}(l_0)$ by l_i^{im}. Since we shall make use of these assumptions over and over again we save time and give them a name

$$\Delta \begin{cases} 0. & l_0\,;\mu_0 \in \mathbf{M}_{sexp} \text{ is such that no cell in } \mathbf{Cells}^{<}_{\mu_0}(l_0) \text{ is marked.} \\ 1. & \Lambda \text{ is the left first spanning tree of } \mathbf{Unmarked}_{\mu_0}(l_0) \text{ at } l_0\,;\mu_0. \\ 2. & \mu = \mathbf{RM}^{\Lambda}_{\mu_0}(l_0). \\ 3. & \mathbf{Cells}_{\mu_0}(l_0) = \{l_0,\ l_1,\ \ldots,l_r\} \\ 4. & (l_i\,;\mu_0)_0 = v_{ia} \text{ and } (l_i\,;\mu_0)_1 = v_{id} \text{ for } i \in r+1. \end{cases}$$

The following proposition is a consequence of our notation.

Proposition 6: $\delta_\mu = \delta_{\mu_0} \cup \{l_0^{im},\ l_1^{im},\ \ldots,l_r^{im}\}$ where

1. $l_i^{im} \neq l_j^{im} \notin \delta_{\mu_0}$ for $i,j \in r+1$ and $i \neq j$
2. $image(l_i)\,;\mu \gg l_i^{im}\,;\mu$ for $i \in r+1$
3. $car(l_i)\,;\mu \gg v\,;\mu$ where $v \in \{\mathtt{M00},\ \mathtt{M01},\ \mathtt{M10},\ \mathtt{M11}\}$ for $i \in r+1$
4. $(l_i\,;\mu_0)_0 = (l_i\,;\mu)_a = v_{ia}$ and $(l_i\,;\mu_0)_1 = (l_i\,;\mu)_d = v_{id}$ for $i \in r+1$.

For convenience we let

$$(v)^{image} = \begin{cases} v & \text{if } v \in \mathbb{A} \\ v & \text{if } v \in \mathbb{L} \text{ but } v \notin \{l_0,\ \ldots,l_r\} \\ l_i^{im} & \text{if } v = l_i \wedge i \in r+1 \end{cases}$$

The main theorem concerning the recursive algorithm is:

Theorem 7: If l_0, μ_0, and μ are as in Δ then

$$rec{:}copy(l_0)\,;\mu_0 \gg l_0^n\,;\mu_1$$

such that

1. $l_0^n \,;\mu_1 \cong l_0 \,;\mu_1$
1. $\mu_1(l_i) = \mu_0(l_i) \quad \text{for} \quad i \in r+1$
1. $\mu_1(l_i^n) = [(v_{ia})^{image}\ (v_{id})^{image}] \quad \text{for} \quad i \in r+1$

We now turn to defining the set upon which induction will be carried out. In this case since the structures that we are working on have a special form, *car-cdr* chains through cells having a certain property are no longer appropriate. What we actually are interested in now is *a-d* chains in the sense of the abstract syntax of the previous section. For this reason we define $\mathbf{TC}_{\{a,d\}}$ in exactly the same way as **TC** except that 0 is replaced everywhere by a and 1 by d. To be precise:

Definition: $\mathbf{TC}_{\{a,d\}}(v\,;\mu,\Psi,\Phi_a,\Phi_d)$ is the smallest set X such that

1. If $\Psi(v\,;\mu)$ then $v \in X$
2. If $l \in X$ and $\Phi_a(l\,;\mu)$ then $(l\,;\mu)_a \in X$
3. If $l \in X$ and $\Phi_d(l\,;\mu)$ then $(l\,;\mu)_d \in X$

The reader is reminded that the definition of $(l\,;\mu)_a$ and $(l\,;\mu)_d$ can be found in 6.1.

Definition of $\mathbf{Tree}_{\mu_t}(v)$: Suppose that μ_t is some memory, it will usually be related to μ but we do not require it, then define

$$\mathbf{Tree}_{\mu_t}(v) = \mathbf{TC}_{\{a,d\}}(v, \mu_t, \Phi_M, \Phi_a, \Phi_d)$$

where $\Phi_a(v\,;\mu) \leftrightarrow (v\,;\mu)_m \in \{\texttt{M00},\ \texttt{M01}\}$, $\Phi_d(v\,;\mu) \leftrightarrow (v\,;\mu)_m \in \{\texttt{M00},\ \texttt{M10}\}$, and $\Phi_M(v\,;\mu) \leftrightarrow \mathit{marked}(v)\,;\mu \gg \mathtt{T}\,;\mu$. In other words

$$\mathbf{Tree}_{\mu_t}(v) = \begin{cases} \emptyset & v \in \mathbb{A} \ \vee \neg\Phi_M(v\,;\mu_t) \\ \{v\} & \text{if } (v\,;\mu_t)_m = \texttt{M11} \\ \{v\} \cup \mathbf{Tree}_{\mu_t}(v_a) & \text{if } (v\,;\mu_t)_m = \texttt{M01} \\ \{v\} \cup \mathbf{Tree}_{\mu_t}(v_d) & \text{if } (v\,;\mu_t)_m = \texttt{M10} \\ \{v\} \cup \mathbf{Tree}_{\mu_t}(v_a) \oplus \mathbf{Tree}_{\mu_t}(v_d) & \text{if } (v\,;\mu_t)_m = \texttt{M00} \end{cases}$$

Notice that $\mathbf{Tree}_{\mu}(l_i) = \Lambda(l_i)$ when $l_i \in \mathbf{Unmarked}_{\mu_0}(l_0)$, for $i \in r+1$.

Definition of $\mathbf{Tree}^*{}_{\mu}(l_i)$: Now if μ is as in Δ and $i \in r+1$ then for convenience we let

$$\mathbf{Tree}^*{}_{\mu}(l_i) = \mathbf{Tree}_{\mu}(l_i) \oplus \{l_j^{im} \mid l_j \in \mathbf{Tree}_{\mu}(l_i)\}$$

7.3. The recursive transformation peel.

We now define the transformation that we use to prove the theorems about peeling. As in the previous section we begin by making explicit the assumptions under which we make this definition. We give them a name so as to refer to them in the future.

Peel condition: Suppose $l_s\,;\mu_t \in \mathbb{M}_{sexp}$ is such that $s \in r+1$ and $\mu_t = \mu$ on $\mathbf{Tree}^*{}_{\mu}(l_s)$. Furthermore if $l_i \prec l_s$ with respect to Λ then $(l_i\,;\mu_t)_1 = l_i^{im}$.

Definition of $\mathbf{Peel}_{\mu_t}(l_s)$: We now define $\mathbf{Peel}_{\mu_t}(l_s)$ a transformation from $\mathbb{M}_{sexp}$ to $\mathbb{M}_{sexp}$. Put

$$\mu^{**} = setcdr(l_s, v_{sd}\,;\, setcar(l_s, v_{sa}\,;\,\mu_t))$$

and

$$\mu^{ad} = setcdr(l_s^{im}, (v_{sd})^{image}\,;\, setcar(l_s^{im}, (v_{sa})^{image}\,;\,\mu^{**}))$$

then

$$\mathbf{Peel}_{\mu_t}(l_s) = \begin{cases} \mathbf{Peel}_{\mathbf{Peel}_{\mu^{ad}}(v_{sd})}(v_{sa}) & \text{if } (l_s\,;\mu_t)_m = \mathtt{M00} \\ \mathbf{Peel}_{\mu^{ad}}(v_{sa}) & \text{if } (l_s\,;\mu_t)_m = \mathtt{M01} \\ \mathbf{Peel}_{\mu^{ad}}(v_{sd}) & \text{if } (l_s\,;\mu_t)_m = \mathtt{M10} \\ \mu^{ad} & \text{if } (l_s\,;\mu_t)_m = \mathtt{M11} \end{cases}$$

Theorem 7. now follows from the following two lemmas.

Rec:peel Lemma: If $l_s\,;\mu_t$ satisfies the Peel condition then

$$rec{:}peel(l_s)\,;\mu_t \gg l_s^{im}\,;\mathbf{Peel}_{\mu_t}(l_s)$$

Proof of *Rec:peel* lemma: This is a simple induction on $|\mathbf{Tree}_{\mu_t}(l_s)|$.

□Rec:peel lemma

Peel Lemma: If $l_s\,;\mu_t$ satisfies the **Peel** condition and if $l_i \in \mathbf{Tree}_{\mu_t}(l_s)$ then

1. $\mathbf{Peel}_{\mu_t}(l_s) = \mu_0$ on $\mathbf{Tree}_{\mu_t}(l_s)$
2. $\mathbf{Peel}_{\mu_t}(l_s)(l_i^{im}) = [(v_{ia})^{image}\ (v_{id})^{image}]$, and
3. $\mathbf{Peel}_{\mu_t}(l_s) = \mu_t$ off $\mathbf{Tree}^*_{\mu_t}(l_s)$.

Proof of Peel lemma: This is by induction on $|\mathbf{Tree}_{\mu_t}(l_s)|$.

Base case: $|\mathbf{Tree}_{\mu_t}(l_s)| = 1$, in this case $(l_s\,;\mu_t)_m = \mathtt{M11}$ and so by proposition 5 l_s is left and right terminal w.r.t Λ. Now $\mu_t = \mu = \mathbf{RM}^{\Lambda}_{\mu_0}(l_0)$ on $\mathbf{Tree}^*_{\mu}(l_s) = \mathbf{Tree}_{\mu_t}(l_s) \cup \{l_s^{im}\}$ so by proposition 6

$$\mu_t(l_s) = [\mathtt{M11}, l_s^{im}] \quad \text{and} \quad \mu_t(l_s^{im}) = [v_{sa}, v_{sd}].$$

Then $\mathbf{Peel}_{\mu_t}(l_s) = \mu^{ad}$ where μ^{ad} is as in the definition of **Peel**. Clearly μ^{ad} has the desired properties. □base case

Induction step: Suppose $|\mathbf{Tree}_{\mu_t}(l_s)| > 1$, and the lemma holds for simpler cases. We shall do the case when $\mu_t(l_s) = [\mathtt{MOO}, l_s^{im}]$, the other two cases being somewhat simpler. In this case we have

$$\mathbf{Peel}_{\mu_t}(l_s) = \mathbf{Peel}_{\mathbf{Peel}_{\mu^{ad}}(v_{sd})}(v_{sa})$$

where again μ^{ad} is as in the definition of **Peel**. Now observe that we may apply the induction hypothesis to v_{sd}. Thus by letting $\mu^* = \mathbf{Peel}_{\mu^{ad}}(v_{sd})$ we have

$$\mu^* = \mu_0 \text{ . on } \quad \mathbf{Tree}_{\mu^{ad}}(v_{sd}) = \mathbf{Tree}_{\mu_t}(v_{sd})$$

and

$$\mu^*(l_i^{im}) = [(l_{ia})^{image}, (l_{id})^{image}] \quad \text{for} \quad l_i \in \mathbf{Tree}_{\mu_t}(v_{sd}).$$

Now again v_{sa} ; μ^* satisfies the hypothesis of the lemma and $\mathbf{Tree}_{\mu_t}(v_{sa}) = \mathbf{Tree}_{\mu^*}(v_{sa})$ is smaller than $\mathbf{Tree}_{\mu_t}(l_s)$ so letting $\mu_f = \mathbf{Peel}_{\mu^*}(v_{sa})$ we have by the induction hypothesis that

$$\mu_f = \mu_0 \quad \text{on} \quad \mathbf{Tree}_{\mu_t}(v_{sa})$$

and

$$\mu_f(l_i^{im}) = [(l_{ia})^{image}, (l_{id})^{image}] \quad \text{for} \quad l_i \in \mathbf{Tree}_{\mu_t}(v_{sa}).$$

Using the definition of μ^{ad} and the fact that

$$\mathbf{Tree}_{\mu_t}(l_s) = \{l_s\} \oplus \mathbf{Tree}_{\mu_t}(v_{sa}) \oplus \mathbf{Tree}_{\mu_t}(v_{sd})$$

we can easily combine the above to give the result. $\square_{\text{Peel lemma}}$

We finish this section with another important property of $\mathbf{Peel}_{\mu_t}(l_s)$, which is proved by an easy induction on $|\mathbf{Tree}_{\mu_t}(l_s)|$.

Commutativity Lemma for $\mathbf{Peel}_{\mu_t}(l_s)$: If l_s ; μ_t satisfies the Peel condition and Γ is a memory operation of the form $\lambda\mu.setx(l_k, v\,; \mu)$ where $x \in \{car, cdr, a, d\}$ and $l_k \in \delta_{\mu_0}$ but $l_k \notin \mathbf{Tree}_{\mu}(l_s)$ then

$$\Gamma(\mathbf{Peel}_{\mu_t}(l_s)) = \mathbf{Peel}_{\Gamma(\mu_t)}(l_s)$$

7.4. The Robson Copying Algorithm

We now present the actual Robson copying algorithm, *copy*. This uses *peel*, a *pointer reversing* version of our *rec:peel* algorithm.

$$copy(\mathtt{s}) \leftarrow \mathtt{if}(atom(\mathtt{s}), \mathtt{s}, peel(rmark(\mathtt{s}), \mathtt{ALPHA}))$$

```
peel(s,stack) ←
    let{nc ← cdr(s),t1 ← a(s),t2 ← d(s)}
       ifs(eq(M00,m(s)),[setm(s,F00),
                         setd(s,stack),
                         peel(t2,s)],
           eq(M01,m(s)),[setm(s,stack),
                         seta(s,t2),
                         setd(s,image(t2)),
                         peel(t1,s)],
           eq(M10,m(s)),[setm(s,F10),
                         setd(s,stack),
                         peel(t2,s)],
           eq(M11,m(s)),[setm(s,t1),
                         seta(s,image(t1)),
                         setd(s,image(t2)),
                         rplacd(s,t2),
                         popstack2(s,stack,nc)])
popstack2(s,stack,newcel) ←
    if(eq(stack,ALPHA),
       newcel,
       let{nc ← cdr(stack),oc ← car(stack),t1 ← a(stack),t2 ← d(stack)}
          ifs(eq(F00,m(stack)),[setm(stack,t2),
                                setd(stack,newcel),
                                seta(stack,s),
                                peel(t1,stack)],
              eq(F10,m(stack)),[setm(stack,t1),
                                seta(stack,image(t1)),
                                setd(stack,newcel),
                                rplacd(stack,s),
                                popstack(stack,t2,nc)],
              T,                [rplacd(stack,t1),
                                setm(nc,newcel),
                                setm(stack,s),
                                popstack2(stack,oc,nc)])
```

The algorithm above uses the abstract syntax defined in 6.1, which for ease of reading, we repeat here.

$m(\mathtt{l}) \leftarrow car(\mathtt{l})$

$a(\mathtt{l}) \leftarrow car(cdr(\mathtt{l}))$

$d(\mathtt{l}) \leftarrow cdr(cdr(\mathtt{l}))$

$setm(\mathtt{l},\mathtt{m}) \leftarrow rplaca(\mathtt{l},\mathtt{m})$

$seta(\mathtt{l},\mathtt{x}) \leftarrow rplaca(cdr(\mathtt{l}),\mathtt{x})$

$setd(\mathtt{l},\mathtt{x}) \leftarrow rplacd(cdr(\mathtt{l}),\mathtt{x})$

$image(\mathtt{l}) \leftarrow \mathtt{if}(atom(\mathtt{l}),\mathtt{l},cdr(\mathtt{l}))$

Where x accesses the *cxr* part of the *image* cell associated with l and $setx$ updates it for $x \in \{a,d\}$. As for the Robson marking algorithm *peel* is defined by a tail recursive system of definitions.

7.5. The Main copying theorem.

The main theorem of this section is

Theorem 8: If l_0, μ_0 and μ are as in Δ then

$$copy(l_0)\,;\mu_0 \gg l_0^{im}\,;\mu_1$$

such that

1. $l_0^{im}\,;\mu_1 \cong l_0\,;\mu_1$
2. $\mu_1(l_i) = \mu_0(l_i)$ for $i \in r+1$
3. $\mu_1(l_i^{im}) = [(v_{ia})^{image}\ (v_{id})^{image}] \underset{\text{df}}{=} [v_{ia}^{im}\ v_{id}^{im}]$ for $i \in r+1$

This is a consequence of the following lemma.

Main Lemma: If $l_s\,;\mu_t$ satisfies the **Peel** condition then

$$peel(l_s,\ v)\,;\mu_t \gg popstack2(l_s,\ v,\ l_s^{im})\,;\mathbf{Peel}_{\mu_t}(l_s)$$

Proof of Main lemma: This is by induction on $|\mathbf{Tree}_{\mu_t}(l_s)|$.

Base case: $|\mathbf{Tree}_{\mu_t}(l_s)| = 1$. In this case $(l_s\,;\mu_t)_m = \mathtt{M11}$ and consequently

$$peel(l_s,\ v)\,;\mu_t \gg popstack2(l_s,\ v,\ l_s^{im})\,;\mu^*$$

where, recalling the definition of μ^{**} and μ^{ad} from the previous section, we have

$$\mu^* = \begin{cases} \mu^{**} & \text{if } v_{sa}, v_{sd} \in \mathbb{A} \\ setcar(l_s^{im}, v_{sa}^{im} ; \mu^{**}) & \text{if } v_{sd} \in \mathbb{A} \wedge v_{sa} \in \mathbb{L} \\ setcdr(l_s^{im}, v_{sd}^{im} ; \mu^{**}) & \text{if } v_{sa} \in \mathbb{A} \wedge v_{sd} \in \mathbb{L} \\ \mu^{ad} & \text{otherwise} \end{cases}$$

By the definition of $(v)^{image}$, in all of the above cases $\mu^* = \mu^{ad}$. $\square_{\text{base case}}$

Induction step: $|\mathbf{Tree}_{\mu_t}(l_s)| > 1$. In this case $(l_s ; \mu_t)_m \in \{\texttt{M00}, \texttt{M01}, \texttt{M10}\}$. Consequently we split this part of the proof into three cases.

Case 1: $(l_s ; \mu_t)_m = \texttt{M00}$. Here we have

$$peel(l_s, v) ; \mu_t \gg peel(v_{sd}, l_s) ; \mu_1$$

where $\mu_1 = setd(l_s, v ; setcar(l_s, \texttt{F00} ; \mu_t))$. Consequently by the induction hypothesis

$$peel(v_{sd},\ l_s) ; \mu_1 \gg popstack2(v_{sd},\ l_s,\ v_{sd}^{im}) ; \mu_2$$

where $\mu_2 = \mathbf{Peel}_{\mu_1}(v_{sd})$. Now since l_s is unchanged during the transformation from μ_1 to μ_2 we have

$$popstack2(v_{sd},\ l_s,\ v_{sd}^{im}) ; \mu_2 \gg peel(v_{sa}, l_s) ; \mu_3$$

where $\mu_3 = seta(l_s, v_{sd} ; setd(l_s, v_{sd}^{im} ; setcar(l_s, v ; \mu_2)))$. Again by our induction hypothesis

$$peel(v_{sa},\ l_s) ; \mu_3 \gg popstack2(v_{sa},\ l_s,\ v_{sa}^{im}) ; \mu_4$$

where $\mu_4 = \mathbf{Peel}_{\mu_3}(v_{sa})$. Since l_s is unchanged we obtain

$$popstack2(v_{sa},\ l_s,\ v_{sa}^{im}) ; \mu_4 \gg popstack2(l_s, v, l_s^{im}) ; \mu_5$$

where $\mu_5 = setcdr(l_s, v_{sd}; setcar(l_s^{im}, v_{sa}^{im}; setcar(l_s, v_{sa}; \mu_4)))$. Now we show $\mu_5 = \mathbf{Peel}_{\mu_t}(l_s)$. Note that $\mu_2 = \mathbf{Peel}_{\mu_1}(v_{sd})$ and $\mu_1 = setd(l_s, v ; setcar(l_s, \texttt{F00} ; \mu_t))$. By the commutativity lemma $\mu_2 = setd(l_s, v ; setcar(l_s, \texttt{F00} ; \mathbf{Peel}_{\mu_t}(v_{sd})))$ and thus

$$\mu_3 = seta(l_s, v_{sd} ; setd(l_s, v_{sd}^{im} ; setcar(l_s, v ; setd(l_s, v ; setcar(l_s, \texttt{F00} ; \mathbf{Peel}_{\mu_t}(v_{sd}))))))$$

which by cancellation this becomes

$$\mu_3 = seta(l_s, v_{sd} ; setd(l_s, v_{sd}^{im} ; setcar(l_s, v ; \mathbf{Peel}_{\mu_t}(v_{sd})))).$$

Now

$$\mu_5 = setcdr(l_s, v_{sd} ; setcar(l_s^{im}, v_{sa}^{im} ; setcar(l_s, v_{sa} ; \mathbf{Peel}_{\mu_3}(v_{sa}))))$$

which by the commutativity lemma becomes

$$\mathbf{Peel}_{setcdr(l_s, v_{sd}; setcar(l_s^{im}, v_{sa}^{im}; setcar(l_s, v_{sa}; seta(l_s, v_{sd}; setd(l_s, v_{sd}^{im}; setcar(l_s, v; \mathbf{Peel}_{\mu_t}(v_{sd})))))))}(v_{sa}).$$

By cancellation, this becomes

$$\mu_5 = \mathbf{Peel}_{setcdr(l_s,v_{sd};setcar(l_s^{im},v_{sa}^{im};setd(l_s,v_{sd}^{im};setcar(l_s,v;\mathbf{Peel}_{\mu_t}(v_{sd}))))))}(v_{sa})$$

and one more application of the commutativity lemma gives the result. $\square_{\text{case 1}}$

Case 2: $(l_s\ ;\mu_t)_m = \texttt{M01}$. In this situation

$$peel(l_s,v)\ ;\mu_t \gg peel(v_{sa},l_s)\ ;\mu_1$$

where

$$\mu_1 = \begin{cases} seta(l_s,v_{sd}\ ;setcar(l_s,v\ ;\mu_t)) & \text{if } v_{sd} \in \mathbb{A} \\ setd(l_s,v_{sd}^{im}\ ;seta(l_s,v_{sd}\ ;setcar(l_s,v\ ;\mu_t))) & \text{otherwise.} \end{cases}$$

Now by induction

$$peel(v_{sa},\ l_s)\ ;\mu_1 \gg popstack2(v_{sa},\ l_s,\ v_{sa}^{im})\ ;\mu_2$$

where $\mu_2 = \mathbf{Peel}_{\mu_1}(v_{sa})$. Since l_s is unchanged

$$popstack2(v_{sa},\ l_s,\ v_{sa}^{im})\ ;\mu_2 \gg popstack2(l_s,v,l_s^{im})\ ;\mu_3$$

where $\mu_3 = setcar(l_s,v_{sa}\ ;setcar(l_s^{im},v_{sd}^{im}\ ;setd(l_s,v_{sd}\ ;\mu_2)))$.

We only show that $\mu_3 = \mathbf{Peel}_{\mu_t}(l_s)$ in the case $v_{sd} \in \mathbb{A}$, the other case is not much more challenging.

$$\mu_3 = setcar(l_s,v_{sa}\ ;setcar(l_s^{im},v_{sa}^{im}\ ;setd(l_s,v_{sd}\ ;\mathbf{Peel}_{\mu_1}(v_{sa}))))$$

So by the commutativity lemma

$$\mu_3 = \mathbf{Peel}_{setcar(l_s,v_{sa};setcar(l_s^{im},v_{sa}^{im};setd(l_s,v_{sd};\mu_1)))}(v_{sa})$$

and since

$$\mu_1 = seta(l_s,v_{sd}\ ;setcar(l_s,v\ ;\mu_t))$$

we have

$$\mu_3 = \mathbf{Peel}_{setcar(l_s,v_{sa};setcar(l_s^{im},v_{sa}^{im};setd(l_s,v_{sd};seta(l_s,v_{sd};setcar(l_s,v;\mu_t)))))}(v_{sa}).$$

By cancellation this becomes

$$\mu_3 = \mathbf{Peel}_{setcar(l_s,v_{sa};setcar(l_s^{im},v_{sa}^{im};setd(l_s,v_{sd};\mu_t)))}(v_{sa})$$

and thus $\mu_3 = \mathbf{Peel}_{\mu_t}(l_s)$. $\square_{\text{case 2}}$

Case 3: $(l_s\ ;\mu_t)_m = \texttt{M10}$. In this situation

$$peel(l_s,v)\ ;\mu_t \gg peel(v_{sd},l_s)\ ;\mu_1$$

where $\mu_1 = setd(l_s, v\,;\, setcar(l_s, \mathtt{F10}\,;\, \mu_t))$. By induction

$$peel(v_{sd},\ l_s)\,;\mu_1 \gg popstack2(v_{sd},\ l_s,\ v_{sd}^{im})\,;\mu_2$$

where $\mu_2 = \mathbf{Peel}_{\mu_1}(v_{sd})$. Furthermore

$$popstack2(v_{sd},\ l_s,\ v_{sd}^{im})\,;\mu_2 \gg popstack2(l_s, v, l_s^{im})\,;\mu_3$$

where

$$\mu_3 = \begin{cases} setcdr(l_s, v_{sd}\,;\, setd(l_s, v_{sd}^{im}\,;\, setcar(l_s, v_{sa}\,;\, \mu_2))) & \text{if } v_{sa} \in \mathbb{A} \\ seta(l_s, v_{sa}^{im}\,;\, setcdr(l_s, v_{sd}\,;\, setd(l_s, v_{sd}^{im}\,;\, setcar(l_s, v_{sa}\,;\, \mu_2)))) & \text{otherwise.} \end{cases}$$

Again we only show $\mu_3 = \mathbf{Peel}_{\mu_t}(l_s)$ in the case $v_{sd} \in \mathbb{A}$. Here we have

$$\mu_3 = setcdr(l_s, v_{sd}\,;\, setd(l_s, v_{sd}^{im}\,;\, setcar(l_s, v_{sa}\,;\, \mu_2))).$$

Using the commutativity lemma and cancelling we obtain

$$\mu_3 = \mathbf{Peel}_{setcdr(l_s, v_{sd}; setd(l_s, v_{sd}^{im}; setcar(l_s, v_{sa}; \mu_t)))}(v_{sd}).$$

□case s

□main lemma

8. Bibliography:

- **A** ... Aho, A. V., Hopcroft, J. E., Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley 1974.
- **B** ... Burstall, R. M.: Some Techniques for Proving Correctness of Programs which Alter Data Structures, in Meltzer, B. and Mitchie, D. (eds.) *Machine Intelligence 7*, Edinburgh University Press, (1972), pp. 23-50.
- **C** ... Steele, G. L.: *Common Lisp* Digital Press, 1984, page 265.
- **D** ... Deutsch, L. P.: p 417, Volume 1, [**K**].
- **F** ... Friedman, H.: *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory.* Logic Colloquium '69, North Holland 1971, pp 316-389.
- **K** ... Knuth, D. E.: *The art of computer programming.* Addison-Wesley, 1968.
- **Mc** ... McCarthy, J.: *Towards a mathematical science of computation.* Information Processing 1962, Proceedings of IFIP Congress 62. pages 21-28.
- **Mo** ... Moschovakis, Y. N.: *Abstract First Order Computability I.* Trans. Amer. Math. Soc. 138, 1969, Pages 427-464.
- **R** ... Robson, J. M.: *A Bounded Storage Algorithm for Copying Cyclic Structures.* Communications of the ACM, June 1977, Volume 20, Number 6, Pages 431-433.

- **S** ... Suzuki, N.: *Analysis of pointer rotation.* Communications of the ACM, May 1982, Volume 25, Number 5, Pages 330-335.
- **SW** ... Schorr, H., and W. M. Waite: *An efficient machine-independent procedure for garbage collection in various list structures.* Communications of the ACM, 1967, Volume 10, Pages 501-506.
- **T** ... Topor, R. W.: *The Correctness of the Shorr-Waite List Marking Algorithm.* Acta Informatica, 1979, Volume 11, Pages 211-221.
- **To** ... Touretzky, D. S.: *LISP: a gentle introduction to symbolic computation.* Harper and Row 1984.
- **Tu** ... Tucker, J. V., et al.: *Finite Algorithmic Procedures and Inductive Definability.* Math Scand. 46. 1980. Pages 62-76.
- **W** ... Wadler, P.: *Listlessness is Better than Lazyness.* 1984 ACM Symposium on Lisp and Functional Programming, Pages 45-53.