The Efficient Implementation of Very-High-Level Programming Language Constructs

R. Hood

August 1982

TR 82-503

Department of Computer Science
Cornell University
Ithaca, New York 14853

THE EFFICIENT IMPLEMENTATION OF VERY-HIGH-LEVEL PROGRAMMING LANGUAGE CONSTRUCTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

Robert Todd Hood

August 1982

THE EFFICIENT IMPLEMENTATION OF
VERY-HIGH-LEVEL PROGRAMMING LANGUAGE CONSTRUCTS

Robert Todd Hood, Ph.D.
Cornell University, 1982

ABSTRACT

An investigation is made into efficiently-implementable very-high-level programming language constructs. In a manner analogous to ALGOL 60's abstraction away from GOTO's, an abstract replacement for pointers (the path) is proposed. The use of paths instead of pointers in unshared recursive data structures greatly simplifies the process of reasoning about programs. The existence of an efficient implementation of paths makes their use palatable as well as desirable. Also investigated is the integration of paths with existing very-high-level programming language implementation techniques, such as hash-consing.

Several real-time programs in Pure LISP are presented. Building on the foundation of a real-time queue and a real-time double-ended queue, a real-time implementation of paths is given. This leads to the surprising negative result that the addition of paths does not increase the "power" of Pure LISP.

BIOGRAPHICAL SKETCH

Robert Hood was born on March 24, 1954 in Wilmington, Delaware. He was graduated from Elkton High School in Elkton, Maryland in 1972. In 1976 he received a Bachelor of Arts degree in mathematics from the University of Virginia, graduating with highest distinction. From 1976 to 1980 he was a graduate student at Cornell University in Ithaca, New York, earning a Masters degree in 1979. In 1980 he took a leave of absence from Cornell to move to Houston, Texas where he worked at Rice University while finishing up his degree requirements.

To my parents

ACKNOWLEDGMENTS

Without the help of my special committee chairman, Corky Cartwright, this thesis would never have been finished. His ideas form the basis for much of the research.

The linear-time queue program of section 4.1 was shown to me by Robert Melville. He and Gary Levin looked over the real-time version and greatly improved my presentation of it. An hour with Alan Demers led to the real-time deque implementation of section 4.2.

I would like to thank David Gries for showing me the light (methodologically speaking.)

I had many interesting discussions with my friends at Cornell. While many of them delayed my finishing, I benefited from all of them.

I don't know if it had a positive effect, but I would be remiss if I did not thank my friends at Rice University who nagged me into finishing. Paul Milazzo was especially persistent.

This thesis was prepared partly on a PDP-11/60 running UNIX† at Cornell University. For his work in system maintainence there I thank Dean Krafft. The remainder of the preparation was done at Rice University on a Vax-11/780 running VMS. Scott Comer does a fine job of maintaining the system there. David Kashtan (formerly formerly of SRI) deserves a great deal of credit for Eunice, a UNIX simulator which runs under VMS. Michael Caplinger and Kenny Zadeck at Rice helped make Eunice usable for the kind of text processing I wanted to do.

†UNIX is a Trademark of Bell Laboratories.

Finally, for their unfailing financial support through grant MCS 78-05850, I would like to thank the National Science Foundation and everybody who paid federal taxes in the years 1978 through 1980.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

A major unsolved problem in the field of programming languages is the development of efficient very-high-level languages — programming languages that support an abstract view of complex data objects, yet execute efficiently. In this thesis we use an approach suggested by Hoare to develop an efficient very-high-level analog of pointers.

Chapter 1 of the thesis discusses the problem in the context of currently available programming languages. In chapter 2 an overview of the proposed solution is presented. For the sake of simplicity, the scope of the solution is limited. Chapter 3 presents a detailed version of the material of chapter 2. Some interesting theoretical results, motivated by the practical goals of our research, are presented in chapter 4. Finally the work of the thesis is summarized and future work is suggested in chapter 5. Three appendices contain detailed examples of material described in the thesis.

1.1. Motivation

The programmer who wants to solve problems in a high-level language is currently forced to choose between efficiency and abstraction in his solution. Conventional high-level programming languages, such as ALGOL 60 [28] and PASCAL [19], have a serious gap between the level of abstraction of control-flow mechanisms and the level of abstraction of data representation. While they hsve liberated programmers from the process of expressing algorithms as sequences of machine instructions,

they still force programmers to represent data objects in terms of machine level primitives. In particular, the efficient representation of complex data objects such as unbounded sequences and sets usually involves pointers and records (or a simulation of pointers and records using arrays). Programs that manipulate pointer and record representations of abstract data objects are difficult to understand and to prove correct because the abstract mathematical structure of the program data domain is obscured by low-level details.

In contrast, a number of very-high-level, pedagogic languages, such as Pure LISP [26] and SETL [21] treat complex data objects as abstract values independent of any particular machine implementation. In these languages, complex data objects such as sequences and sets are ordinary data values just like integers. The underlying machine implementation consisting of pointers and records is invisible to the programmer. Consequently, it is much easier to reason (both formally and informally) about programs manipulating complex data objects. From a practical standpoint, the fatal flaw in very-high-level languages is inefficiency of execution. Not surprisingly, the major source of the inefficiency in these languages is the absence of "impure" data operations that directly manipulate data representations.

A promising approach to the problem of developing efficient very-high-level languages is to follow the course that the designers of ALGOL used to gradually supplant the **go to** statement: invent a collection of abstract constructs to perform the same functions as the common disciplined uses of the low-level construct. In fact, C. A. R. Hoare [17] has observed,

> There appears to be a close analogy between references in data and jumps in a program. A jump is a very powerful multipurpose tool present in the object code produced by compilers for almost every machine. But it is also an undisciplined feature, which can be used to create wide interfaces between parts of a program which appear to be disjoint. That is why a high level programming language like ALGOL 60 has introduced a range of program structures such as compound statements, conditional statements, while statements, procedure statements, and recursion to replace many of the uses of the machine code jump. Indeed perhaps the only remaining purpose of the jump is to indicate irreparable breakdown in the structure of the program. Similarly, if references have any role in data structuring it may be a purely destructive one. It would therefore seem highly desirable to attempt to classify all those special purposes to which references may be put, and replace them in a high level language by more structured principles and notations.

In the language implementation, the abstract constructs can be compiled or interpreted in essentially the same way as their low-level equivalents. The only necessary change is the addition of the machinery required to enforce the constraints on the constructs' use.

By employing this strategy, we have developed a computationally efficient abstract alternative to the most common disciplined use of impure operations in LISP: selectively updating (and destructively modifying) unshared data objects. Although we initially discuss the new construct in the limited context of manipulating LISP S-expressions, our ideas easily generalize to arbitrary recursive data structures as defined by McCarthy [27], Hoare [17], and Cartwright [5].

1.2. Current Work in High-level Languages

The major contribution of high-level languages has been the introduction of control-flow mechanisms that greatly ease program

development. Work done by Floyd [13], Hoare [15], and Dijkstra [12] has made correctness proofs feasible for some programs. The major deficiency of such languages is the absence of data abstraction mechanisms. The programmer who wants to use a binary tree, for example, is forced to implement it himself using arrays or pointers and records.

Because of the nature of pointers and records (or arrays used to simulate them) proofs of correctness of programs using them are extremely complicated. Understanding a PASCAL assignment statement such as

$$p\uparrow := x \qquad (1.2.1)$$

requires knowledge of the entire record class of p, together with all pointers belonging to that class. That is, the dereferenced pointer p is a possible alias for any other dereferenced pointer belonging to the same class, and the semantics of (1.2.1) must take that into account.

To illustrate how unwieldy proofs involving pointers and records can be, we will examine a program and proof by Luckham and Suzuki [25]. The program (reproduced in figure 1.2.2) implements the search and increment operation of an event queue. The event queue is a linear list of records where each record consists of a key field, a count field, and a next field. Variable Root points to the first cell, and Sentinel to the last (a dummy cell).

After presenting the problem and solution, Luckham and Suzuki proceed with their proof of correctness. Their proof, actually a proof of loop-freeness of their implementation, depends on 14 axioms and 20 lemmas (which are asserted as axioms). Even though the use of pointers

```
type Ref  = ↑Word;
     Word = record Key   : integer;
                   Count : integer;
                   Next  : Ref end;
procedure Search( X : integer; Sentinel : Ref; var Root : Ref);
var W1, W2 : Ref;
begin
   Sentinel↑.Key := Next;
   if W1 = Sentinel then
   begin
      NEW(Root);
      Root↑.Key := X; Root↑.Count := 1; Root↑.Next := Sentinel;
   end else
   if W1↑.Key = X then W1↑.Count := W1↑.Count+1 else
   begin
      repeat W2 := W1; W1 := W2↑.Next
        until W1↑.Key = X;
      if W1 = Sentinel then
      begin
         W2 := Root; NEW(Root);
         Root↑.Key := X; Root↑.Count := 1; Root↑.Next := W2
      end else
      begin
         W1↑.Count := W1↑.Count + 1;
         W2↑.Next  := W1↑.Next;
         W1↑.Next  := Root; Root := W1
      end
   end
end;.
```

Figure 1.2.2
Luckham and Suzuki's event queue

in the program is straightforward, the pathological nature of pointers causes an inordinately long proof.

1.3. Previous Work in Very-high-level Languages

It is unfortunate that one of the most high-level programming languages available is noted primarily for its inefficiency of execution. A discussion with anyone who has used SETL will lead one to believe that the use of the language is inappropriate for all but the

smallest problems. SETL programs run only 3% as fast as corresponding FORTRAN programs. The chief source of the inefficiency is the inability of the programmer to control how abstract data objects are implemented. If the compiler makes a poor representation choice, the program runs slowly because the critical operations are not efficiently implemented.

Some languages that support a rich abstract-data domain, such as LISP, also provide mechanisms for subverting the abstraction for efficiency's sake. The LISP functions rplaca and rplacd directly modify the pointer and record structures representing S-expressions. The semantics of these operations cannot be described at the abstract level of S-expressions; they have meaning only at the level of the underlying implementation.

Despite the logical complexity of impure operations, they are indispensable in many practical applications because they enable the programmer to write much more efficient programs. As an illustration, consider the problem of maintaining a queue in Pure LISP. The standard solution is to store the queue as a linear-list, inserting new elements at the end and removing elements from the front. The operations of inserting and removing the first element require only constant time. Inserting an element at the end of the list, however, requires time proportional to the length of the list. By using more sophisticated data structures and algorithms, it is possible to reduce the asymptotic time bound, but the resulting programs are much more complicated.[1]

[1] See chapter 4.

On the other hand, if we allow impure (destructive) operations, we can improve the efficiency of the simple linear list solution so that all operations take only constant time. The modification is obvious: maintain a pointer to the last record of the list representation and use the pointer to destructively update the list (using xplacd) when inserting a new element. It is not only more efficient than the Pure LISP solutions, but it seems logically simpler as well. Nevertheless, it is difficult to prove that the impure solution actually implements a queue. The Pure LISP solutions have simpler proofs because they are expressed at a much higher level of abstraction.

Kieburtz [20] has also recognized the destructive nature of pointer operations. He recommends the use of recursive data structures and auxiliary value variables to replace pointers. An auxiliary value variable is bound to a location within a larger value and can be used to select a substructure. While the approach has some merits, it is not a general solution to the problem.

1.4. A Note on Notation

Throughout the thesis we will use the convention of putting programming language keywords in **bold face** type. To distinguish them from ordinary written text, simple variables and function names appearing in text will be underlined. Composite designations of variables such as "v.p", however, will not be underlined, since there can be no ambiguity. In addition, variables and function names in figures will be in Roman.

CHAPTER 2

AN OVERVIEW OF THE RESEARCH

Although pointers have pathological semantics, in general, they are frequently used in a very disciplined way: to update variables with unshared values — i.e. values represented by pointer-and-record structures that are not part of the representations of any other values. In this situation, a pointer serves as an abstract index, analogous to an array index, identifying a particular location within the variable's value. In the context of very-high-level value-oriented languages such as LISP, impure pointer-based operations provide an efficient mechanism for replacing the structure at a specified location within a variable by a new structure, thereby modifying the value of the variable. In this limited context, low-level destructive operations have a simple abstract interpretation: they update a value at an abstract location within the value.

In this chapter we will develop an abstract, yet efficient, alternative to using pointers as generalized indices into LISP S-expressions.[1] As the first step in this process, we must formalize the intuitive concept of "abstract locations" and create operations for manipulating them.

[1]For the sake of simplicity, we limit the investigations of this chapter to one recursive data structure (S-expressions). A more extensive proposal is described in Chapter 3.

2.1. Paths as Abstract Locations

A natural description for a location within a recursive data structure (such as an S-expression) is a sequence of selector names specifying the path from the root to the location. We call such a sequence a *path*. In accordance with typical notation for sequences, we denote the path consisting of the sequence of selector names $s_1, s_2, \ldots, s_n$ by $\langle s_1, s_2, \ldots, s_n \rangle$. The empty path is denoted by $\langle\rangle$. In the case of LISP S-expressions, the elements of the path are taken from the two element selector set {CAR, CDR}.[2] For example, in the list (A B C) the path <CDR, CAR> refers to the second element, B, while the path <CDR> refers to the tail (B C).

Like array indices, paths can be used either to extract values from composite values or to update them. The function

$$\text{extract} : \text{S-expr} \times \text{path} \rightarrow \text{S-expr}$$

extracts values from within S-expressions; the function

$$\text{update} : \text{S-expr} \times \text{path} \times \text{S-expr} \rightarrow \text{S-expr}$$

replaces values within S-expressions. For example,

```
extract((A B C), <CDR>) = (B C)
extract((A B C), <CDR, CAR>) = B
extract((A B C), <CDR, CDR>) = (C)

update((A B C), <CDR, CAR>, D) = (A D C)
update((A B C), <CDR>, (A)) = (A A)
update((A B C), <CDR, CDR, CDR>, (D)) = (A B C D)
```

[2] The selector names CAR and CDR are capitalized to distinguish them from the selector functions *car* and *cdr*. In other words, capitalization serves as our quoting convention.

The two functions are defined as follows:

$$\{ \text{ return the value in v at the end of path } \langle s_1, \ldots, s_n \rangle \}$$
$$\text{extract} (\text{ v} : \text{S-expr}, \ \langle s_1, \ldots, s_n \rangle : \text{path})$$
$$\equiv \begin{cases} \text{v} & \text{if } n = 0 \\ \text{extract}(\text{car}(\text{v}), \langle s_2, \ldots, s_n \rangle) & \text{if } s_1 = \text{CAR} \\ \text{extract}(\text{cdr}(\text{v}), \langle s_2, \ldots, s_n \rangle) & \text{if } s_1 = \text{CDR} \end{cases}$$

$$\{ \text{ return the value of v with the value at the } \}$$
$$\{ \text{ end of } \langle s_1, \ldots, s_n \rangle \text{ replaced by new\_part } \}$$
$$\text{update} (\text{ v} : \text{S-expr}, \ \langle s_1, \ldots, s_n \rangle : \text{path}, \text{ new\_part} : \text{S-expr})$$
$$\equiv \begin{cases} \text{new\_part} \quad \text{if } n = 0 \\ \text{cons}(\text{update}(\text{car}(\text{v}), \langle s_2, \ldots, s_n \rangle, \text{new\_part}), \\ \quad \text{cdr}(\text{v})) \quad \text{if } s_1 = \text{CAR} \\ \text{cons}(\text{car}(\text{v}), \\ \quad \text{update}(\text{cdr}(\text{v}), \langle s_2, \ldots, s_n \rangle, \text{new\_part})) \quad \text{if } s_1 = \text{CDR} \end{cases}$$

Note that *update* is an ordinary function; it does not modify the value *v* as a side effect.

We now define three functions for manipulating paths: *extend*, *last*, and *retract*. They correspond to the standard *cons*, *car*, and *cdr* operations on S-expressions except that they operate on the tail of a sequence rather than the head.

$$\text{extend}(\langle s_1, \ldots, s_n \rangle : \text{path}, \ f : \text{field-sel})$$
$$\equiv \langle s_1, \ldots, s_n, f \rangle$$

$$\text{last}(\langle s_1, \ldots, s_n \rangle : \text{path})$$
$$\equiv s_n$$

$$\text{retract}(\langle s_1, \ldots, s_n \rangle : \text{path}) \quad (\text{abbreviated } \uparrow\langle s_1, \ldots, s_n \rangle)$$
$$\equiv \langle s_1, \ldots, s_{n-1} \rangle$$

For the sake of convenience, we also define a concatenation operator "$\downarrow$"[3], which behaves much like the append function in LISP:

$$\langle s_1, \ldots, s_n\rangle \downarrow \langle t_1, \ldots, t_m\rangle = \langle s_1, \ldots, s_n, t_1, \ldots, t_m\rangle.$$

The concatenation operator (append) is recursively definable in terms of extend, last, retract, and equality; the definition is essentially identical to the usual recursive definition of LISP append:

```
p ↓ q = if q = <>
          then p
          else extend(p↓(↑q), last(q))
        fi
```

To support imperative programming where variables and assignment are present, we introduce a path selection mechanism analogous to array selection in conventional languages. Modeling our notation after array selection in ALGOL, we let

$$v[p] \tag{2.1.1}$$

specify the "global" location (within the entire program state) at the end of path *p* in variable *v*. Adopting the terminology of Kernighan and Ritchie [22], we call expressions denoting global locations *l-values*. We also adopt the usual convention that l-values are implicitly dereferenced (coerced) in "right-hand" contexts (e.g. within an expression). As a result, path selections may freely appear in both left and right-hand contexts just like array selections. In other words, a path selection such as in (2.1.1) can be used within an expression to extract the

[3]We use a down-arrow ($\downarrow$) for the append operator, because $p_1 \downarrow p_2$ is p_1 extended downward by p_2. We visualize an S-expression as a tree that grows down from its root at the top.

value at location *p* within *v*, and as the target (left-hand-side) of an assignment statement to update *v* at location *p*. For example,

$$v_1[p_1] := v_2[p_2] \tag{2.1.2}$$

is equivalent to the simple assignment

$$v_1 := \mathrm{update}(v_1, p_1, \mathrm{extract}(v_2, p_2)).$$

Like array selection and modification, statement (2.1.2) has a straightforward Dijkstra-style predicate transformer (see section 3.2).

2.2. Restrictions to Facilitate Efficient Implementation

To implement paths efficiently, we must impose some restrictions on their use. In this section, we describe a feasible set of restrictions, and describe some useful operations that have particularly efficient implementations.

With any variable of type S-expression the programmer can declare a number of "built-in" paths that are associated only with that variable. Value variables together with their built-in paths will be called **struct** variables. More precisely, a **struct** variable s consists of a value part and a set of path variables. The association of paths with a value variable is made in a declaration similar to a PASCAL **record** declaration. The statement

$$\textbf{var}\ s\ \textbf{struct}\ v\ \textbf{with}\ p_0 \ldots p_n; \tag{2.2.1}$$

declares a **struct** *s* whose value part is s.v and whose built-in paths are $s.p_0, \ldots, s.p_n$.

To avoid cumbersome notation for l-values involving built-in paths, we use the abbreviation

$s.p_i!$

in place of "$s.v[s.p_i]$" to denote the l-value at the end of $s.p_i$ within s.v. Similarly, we will use

$s.p_i{\downarrow}q!$

to denote the l-value at the end of $(s.p_i{\downarrow}q)$ within s.v. In both cases, we allow "$s.p_i$" to be abbreviated "p_i" if it is unambiguous.

To accommodate fast accessing and updating of S-expressions through built-in paths, we force every **struct** _s_ to obey the following: every built-in path must specify a valid location within the value. We enforce this condition by imposing the following restrictions on updating **struct** _s_. A _speared node_ in _s_ is any node — except the last one — on any built-in path within the value.[4] If all built-in paths are empty, for example, then no node in the value is speared. To preserve the condition, _we prohibit assignments that replace speared nodes_. Similarly, we prohibit assignments to either paths or the entire structure that falsify the condition. In particular, extensions to a path that are not valid in the current value of the variable are prohibited.

The above restrictions are easy to enforce and guarantee that the invariant is preserved. Assignments to the value part that replace speared nodes can falsify the invariant because they modify the structure traversed by a built-in path. For example, if s.v = (A B C) and the

[4] They are called _speared_ because the path impales them. The path goes _to_ but not _through_ the last one; hence it is not speared.

built-in path s.p = <CDR, CDR, CDR>, (i.e. s.v[s.p] = NIL) then the assignment

s.v[<CDR>] := (A)

would yield s.v = (A A), but s.p would be invalid because cdr(cdr(cdr(A A))) does not exist. The most straightforward solution to this problem is to ban the potentially offending assignments. We will discuss possible liberalizations of this policy in Chapter 3.

With the addition of built-in paths, our implementation includes three basic kinds of paths:

(1) path constants — sequences of selectors using the angle-bracket notation as in $<s_1, \ldots, s_n>$. Path constants can be given names by declaring them with a **const** attribute.

(2) built-in paths — paths associated with a **struct** variable _s_.

(3) general path variables — variables whose values are sequences of selectors.

At the semantic level, of course, there is no distinction between the three kinds of paths.

In general, operations involving built-in paths can be performed much more efficiently than the corresponding general path operations. Fortunately, built-in variables seem to suffice for most situations that arise in practice. Moreover, in many cases (such as the queue example) the path is conceptually an integral part of the data structure, making the use of built-in paths particularly appropriate. For the sake of generality, we believe that paths independent of particular S-expression variables should be available in the programming language, but the programmer must be warned that their implementation is costly in comparison to built-in paths.

Two useful path operations, shrinking and growing, have particularly efficient implementations in the context of built-in paths. Shrinking is an operation that replaces a **struct** variable by a substructure of itself. In this operation, the paths bound to the structure are truncated at the front so that they refer to the same value within the structure after the operation as before. (A shrinking operation is prohibited if all the built-in paths do not have the same initial field.) For s a **struct** v **with** $p_0 \ldots p_n$, shrink is defined as follows:

```
shrink(s : struct v with p0 ... pn) ≡
    s.v  := s.v[<f1>];
    s.pi := <f2, ..., fm>
      for all paths pi in {p0, ..., pn}
         where pi has old value <f1, ..., fm>.
```

Growing replaces a **struct** variable by a superstructure of itself. As with shrink, the built-in paths are modified so that they refer to the same value after the growing operation as before. Where s is a **struct** v **with** $p_0 \ldots p_n$, grow is defined as follows:

```
grow(s : struct v with p0 ... pn, f : field-sel, newval : S-expr) ≡
    s.v  := update(newval, <f>, s.v);
    s.pi := <f>+pi
       for all paths pi in {p0, ..., pn}
```

Note that the value newval[<f>] has no effect on the outcome of the operation; it is simply a placeholder that is filled by the old value of s.

2.3. Two Examples

To illustrate the use of built-in paths, we present two simple programming exercises expressed in a PASCAL-like language, where the control structures have been replaced by guarded commands and the data types have been extended to include **struct** variables and path operations.

Figure 2.3.1 shows a solution to the queue problem discussed earlier. A queue q is declared as

var q : **struct** items **with** end

where items is a list of the queue elements, head first. A path to the NIL value at the end of q is kept in the built-in path end. (Path end is a sequence of 0 or more CDR's.) A proof of correctness of this queue implementation appears in a previous paper [9].

```
var q struct items with end; {initial values NIL, <>}

{return the first element of q}
function Qfront(q : struct items with end)
   return car(q.items)
   end Qfront;

{delete the first element of q}
procedure Qdelete(var q : struct items with end)
   shrink(q)
   end Qdelete;

{add v to the end of q}
procedure Qinsert(var q : struct items with end, v : S-expr)
   q.end! := cons(v, NIL);
   q.end := q.end + <CDR>
   end Qinsert
```

Figure 2.3.1
A Solution to the Queue Problem

The second example is paths implementation of the event queue program of Luckham and Suzuki [25]. Their solution (in PASCAL) appears in Figure 1.2.2. As we pointed out in Chapter 1, they prove (with substantial machine assistance) the loop-freeness of the event queue, but stop short of proving that their implementation correct. An event queue using paths is implemented in Figure 2.3.2 and proved correct in Appendix A.

```
Search(var q : s-expr, x : int)
    var t : s-expr, p : path;

    p := <>;
    do q[p] ≠ NIL ∧ q[p↓<CAR, CAR>] ≠ x →  p:= p↓<CDR> od;
    if q[p] = NIL →  t:= cons(x, 0)
     □ q[p] ≠ NIL →  t:= q[p↓<CAR>];q[p]:= q[p↓<CDR>]
    fi;
    q:= cons(t, q);
    q[<CAR, CDR>]:= q[<CAR, CDR>] + 1
    end Search
```

Figure 2.3.2
An Event Queue (after Luckham and Suzuki [25]) using Paths

2.4. Implementation

In discussing the implementation of paths we will initially restrict our attention to the special case of **struct** S-expression variables with a single built-in path.[5] We represent recursive data structures by standard pointer-and-record list structures including reference count fields for storage management. With the exception of a single

[5] Section 3.4 describes modifications for liberalizing these restrictions.

Boolean flag (the "is_speared" flag described at the end of this section), no additional fields are required to support the implementation of paths. Since flag bits are typically available in the low order bits of record pointers, there is no space penalty — beyond the space required for reference counts — involved in supporting paths.[6]

An efficient implementation of paths must perform primitive path operations (with the exception of a few pathological cases) in time independent of path length. Otherwise, algorithms expressed in terms of path operations will be asymptotically less efficient than the corresponding algorithms employing impure pointer operations. For example, if adding an element **v** to the end of a queue **q**,

q.end↓:= cons(v, NIL),

takes time proportional to the length of the built-in path **end**, then the queue implementation presented in Figure 2.3.1 is no faster (asymptotically) than the most straightforward Pure LISP implementation. On the other hand, if the assignment is performed in constant time, then the path solution matches the asymptotic efficiency of the pointer solution.

To perform path operations in constant time, we need a representation of paths that allows us to extract and update list structures

[6] Unless the path implementation is micro-coded, the time cost involved in using low order bits as flag bits may be prohibitive. (Address calculations might be forced to "and" out the low order bits.) A micro-coded implementation, however, would not be impacted too greatly by such computations.

without traversing them. The obvious solution is to represent the path <u>p</u> of

var s **struct** v **with** p

as a pointer to s.p!. Our implementation employs a slight variant of this approach. Since an <u>update</u> must change a field in the <u>father</u> of s.p!, a built-in path representation consists of the pair

[pointer to the father, field selecting the son] .

For example, suppose

s.v = (A (B C))

and

s.p = <CDR, CAR, CDR>

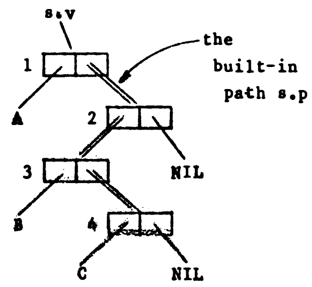
as illustrated in Figure 2.4.1. S.p is represented by the pair [pointer to node 3, CDR]. After execution of

s.p! := (D E) ,

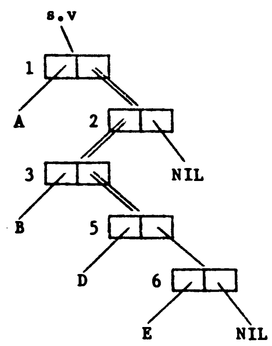
the CDR field of node 3 is changed to point to (D E) (i.e. node 5). S.p remains unchanged.

We call the last node speared by a built-in path (node 3 in the example in Figure 2.4.1), the <u>target</u> node of the path. Since the address of the target node is stored in the path representation, we say that a built-in path <u>points to</u> its target node.

To implement assignment efficiently and to utilize space effectively, an implementation must avoid recopying list structure whenever possible. On the other hand, it must also ensure that updating a



(a) before s.p! := (D E)



(b) after s.p! := (D E)

Figure 2.4.1: Implementation of s.p! := (D E)

variable sharing list structure does not generate side effects (modify the other variables sharing the structure). In the context of destructively updating a structure at the end of a built-in path, the implementation must verify that the speared nodes are not shared. Since an update algorithm cannot explicitly check the appropriate reference counts in constant time, we avoid the problem by prohibiting sharing of speared nodes in the first place. In the implementation, we require that all path operations maintain this invariant. In particular, <u>extend</u> must recopy the new target node if it is shared. Since speared nodes are never shared, the implementation can perform an update by destructively modifying the target node. As a result an update operation usually takes only constant time. The only exception occurs in the unlikely situation when the new component's root is itself a speared node. In this case, the update operation must recopy all the speared nodes of the new component. Since each node has an "is_speared" bit, it is trivial to check whether a node is speared.

To implement the cons operation efficiently, we employ a reference counting scheme for storage management instead of garbage collection. Although Baker [3] has shown that real-time garbage collection is feasible, reference counting is a simpler, more elegant solution in the case of path operations. Since all destructive operations on variables involving paths have semantic definitions in terms of non-destructive operations (pure functions and simple assignment), it is impossible to form circular list structures. Consequently, a pure reference counting scheme can manage storage effectively without recourse to garbage collection. In fact, reference counting seems ideal for this situation because some form of reference count information is necessary to implement paths efficiently. Without reference counts, we cannot tell whether a particular piece of list structure is shared, precluding the implementation of <u>update</u> by destructively modifying the target node.

In a conventional reference counted storage management system, freeing a node can force the reclamation of an unbounded number of nodes (all descendants of the freed node that become inaccessible after the node is freed). Hence, any operation that frees a node has unbounded running time. Fortunately, there is a simple modification to conventional reference-count storage management (suggested by Baker [3]) that evenly distributes the cost of storage reclamation over subsequent node allocations. The modified scheme, called <u>lazy free-list management</u>, does not propagate the decrementing of reference counts generated by freeing a node until the node is reallocated.

Since the reference count of a speared node is 1, we can use the reference count field of speared nodes to hold a back pointer. The back pointer supports the efficient implementation of the retraction operation.

In order to facilitate the implementation of paths in this chapter, several restrictions were placed on their use:

(1) The value part of a **struct** variable must be an S-expression.

(2) Extensions to a built-in path that are not valid in the current value are prohibited.

(3) Only one path can be associated with a **struct** variable.

(4) Update operations cannot replace speared nodes.

(5) Path variables must be associated with (i.e. "built-into") a particular **struct** variable.

For the most part, these restrictions can be relaxed substantially. A more general implementation is described in section 3.4.

CHAPTER 3

PRACTICAL INVESTIGATIONS

In this chapter we develop the design and implementation of a practical formulation of paths. The first section describes features desirable for an efficient very-high-level programming language. The second section sets forth the semantics of paths. In the third section a large example, a program synthesizer, is given. The fourth section fills in the some of the details of the path implementation outlined in section 2.4. The final section assesses the efficiency of the path implementation.

3.1. Language Constructs

The designer of a very-high-level programming language must be concerned with its efficiency of execution as well as its level of abstraction. Inefficiently implemented constructs, no matter what their level of abstraction, should be avoided if the designer expects a wide degree of applicability of his language. Therefore, in this section we limit our discussion to those constructs that have efficient implementations.

The exact choice of control-flow constructs in a programming language is largely a matter of personal preference. We will use Dijkstra's guarded commands for iteration and alternation [12]. The only modification desired is the addition of an **otherwise** guard for the **if...fi** construct. Semantically **otherwise** is equivalent to the conjunction of the negation of the other guards. (Of course, only one **otherwise** clause per statement is permitted.) The inclusion of the

otherwise clause makes the formulation of programs such as the one in Figure 3.3.5 much clearer.

For a procedure call mechanism we use call-by-reference parameter passing with aliasing prohibited. Although the semantics of aliasing are understood [11], the absence of aliasing makes reasoning about procedure calls easier. In addition to procedures, we allow functions that have no side effects.

The organization of data is a major consideration in the design of a programming language. It is important, from the standpoint of verification for data objects to have simple, abstract mathematical properties. It is also important, however, for data objects and operations to have efficient implementations. We propose using standard atomic data types (e.g. **integer**, **boolean**, **character**, etc.) together with well-defined ways of constructing more complicated composite objects (e.g. **array**, recursive types, etc.).

To provide a mechanism for user-defined recursive data structures we will use the constructive data types of Cartwright [5]. He allows declarations of the form:

```
constructor atom(),
            ϵ-seq(),
            seqcons( first:atom,rest:seq);
union seq = ϵ-seq ∪ seqcons
```

which declares a data type <u>seq</u> that behaves in a manner similar to a list of atoms in LISP.

Two critical constructs for abstraction are **set** and **map**. Although both constructs can be described within the framework of Cartwright's constructive data type mechanism, it is desirable (for efficiency's sake) for them to be special cases. For example, we propose using a hashing mechanism (which is described more fully in section 3.4.) to implement maps.

For efficiency's sake, we include the paths described in chapter 2. Instead of restricting their scope to S-expressions, we allow them to be used in arbitrary recursive data structures. The declaration of **struct** variables is modified slightly from its form of (2.2.1). The statement

var s : **struct** v : seq **with** p

declares a **struct** variable s with a built-in path s.p, whose value part, s.v, is of type seq.

3.2. Reasoning about Programs with Paths

Since paths are ordinary sequences and the path operations described in section 2.1 (extract, update, extend, last, and retract) are functions without side effects, it is straightforward to develop a simple first-order axiomatization — including a structural induction schema — for a data domain consisting of paths and recursive data structures. Cartwright [5] describes how to generate such an axiomatization. Within this formal system, we can treat recursive function definitions (as in Pure LISP) over the data domain as logical definitions extending the system [10].

Using this approach, we can prove theorems about paths and recursive data structures by using natural structural induction arguments like those in the Boyer-Moore LISP theorem prover [4], and the Stanford

Typed Lisp verifier [6] [7]. In Appendix A we illustrate the approach in the proofs of two lemmas used in a program correctness argument.

To reason about imperative programs (programs involving assignment) using paths, it is sufficient to apply Hoare-style proof rules [15] or Dijkstra's predicate transformers [12] to reduce the correctness of an asserted program to the truth of a collection of sentences in the first-order system described above. The only extension required is the development of proof rules or predicate transformers to handle explicit and implicit (e.g. shrink) path-indexed assignment statements. To solve the problem, we simply treat path-indexed assignments like indexed assignments to ordinary arrays [11] [14]. The predicate transformer substitutes an entire updated value for the aggregate variable on the left hand side. For example, the predicate transformer suggested by Gries and Levin [14] corresponding to the array element assignment

a[i]:= E

is

wp("a[i]:= E", R(a)) = R(a ; i : E)

where the expression "(a; i: E)" stands for an array identical to a except that the i^{th} element is E.

The exact form of the predicate transformer for path-indexed assignments depends on the particular restrictions placed on paths by the language implementation. We will restrict our attention to general and built-in paths in the context of S-expressions.

In order to guarantee that a dereferenced path refers to a legitimate value, we define a predicate whose purpose is analogous to array subscript checking. Let the function

valid: paths × S-expressions → Boolean

be defined function as follows:

```
valid(p, v) ≡
    if p=<> then true
    else if v ≠ () then false
    else valid(tail(p), head(p)(v))
```

where head and tail are car and cdr operations for sequences. Explicit path assignments to ordinary variables obey the predicate transformer

$$wp(\text{"}v[p]:= E\text{"}, R(v)) = valid(p, v) \wedge R(update(v, p, E)) \qquad (3.2.1)$$

where update is the function defined in section 2.1. In the notation of Gries and Levin, equation (3.2.1) would be written as follows:

$$wp(\text{"}v[p]:= E\text{"}, R(v)) = valid(p, v) \wedge R((v\ ;\ p : E))$$

where (v ; p : E) stands for a value the same as v except that the value at abstract location p has been replaced by E. (v ; p ; E) is recursively defined as follows:

$$(v\ ;\ <> : E) \equiv E$$

$$(v\ ;\ <r>{\downarrow}p_1 : E)[<t>{\downarrow}p_2] \equiv \begin{cases} r = t \rightarrow & v[<t>{\downarrow}p_2] \\ r \neq t \rightarrow & (v[<t>]\ ;\ p_1 : E)[p_2] \end{cases}$$

Implicit path-indexed assignments performed by procedures shrink and grow are treated in the same way as their explicit counterparts.

Each procedure is equivalent to a simple assignment statement. The call

shrink(s)

where s is a **struct** v **with** $p_0 \ldots p_n$, is semantically identical to the assignment

$$s := struct(s.v[head(s.p_0)], tail(s.p_0), \ldots, tail(s.p_n))$$

where struct is a constructor that forms a structure object from a value and a list of paths. It is implicit in the definition of shrink(s) that all paths associated with s have the same first field selector.

Similarly,

grow(s, f, t)

where s is a **struct** v **with** $p_0 \ldots p_n$, t is an S-expression, and f is a field-selector, is equivalent to:

$$s := struct(update(t, f, s.value), <f>{\downarrow}s.p_0, \ldots, <f>{\downarrow}s.p_n) .$$

Hence, if s is a **struct** v **with** $p_0 \ldots p_n$, the predicate transformers for shrink and grow are:

$$wp(shrink(s), R(s)) = \forall 0<i, j<n(head(p_i) = head(p_j) \wedge p_i \neq <>) \wedge R(struct(s.v[head(s.p_0)], tail(s.p_0), \ldots, tail(s.p_n)))$$

$$wp(grow(s, f, t), R(s)) = \neg atom(t) \wedge R(struct(update(t, f, s.v), <f>{\downarrow}s.p_0, \ldots, <f>{\downarrow}s.p_n)) .$$

To demonstrate how easy it is to reason about paths, in Appendix A we prove the correctness of the event queue of Figure 2.3.2.

3.3. A Large Example--A Program Synthesizer

In this section we implement a simple program synthesizer — an editor directed by the syntax of a programming language (see Teitelbaum [32]). Input to the synthesizer is a sequence of commands that move a cursor, and insert or delete statements. For the sake of simplicity, the language consists only of assignment statements and Dijkstra's guarded IF and DO statements. It is described by the grammar in Figure 3.3.1.

The user of the synthesizer adds text to his program by typing statements and expressions and by inserting statement templates. Assignment statements and the boolean expressions for guards are typed in directly (and then parsed by the synthesizer); **do** and **if** statements, however, are added using templates. To illustrate what the user of the synthesizer sees, assume the initial program is

```
<stmt>
```

where the position of the cursor is indicated by highlighting (in a box)

```
<program>  ::= [<stmtlist>]
<stmtlist> ::= <stmt> { ; <stmt> }
<stmt>     ::= skip | abort | <assign> | <dostmt> | <ifstmt>
<assign>   ::= <target> := <value>
<dostmt>   ::= do <gcmd> { □ <gcmd> } od
<ifstmt>   ::= if <gcmd> { □ <gcmd> } fi
<gcmd>     ::= <boolean-expression> → <stmtlist>
```

Figure 3.3.1
Grammar of synthesized language

the program construct selected. After performing the command INSERT "i:= 0" the display shows

```
i:= 0;
<stmt>
```

After inserting a "DO" template at the cursor, the program is displayed as follows:

```
i:= 0;
do <guard> →  <stmt>
 □ <guarded cmd>
od
<stmt>
```

Moving the cursor "IN" once positions it so that it selects the first guarded command:

```
i:= 0;
do <guard> →  <stmt>
 □ <guarded cmd>
od
<stmt>
```

Moving it "IN" again positions it as follows:

```
i:= 0;
do <guard> →  <stmt>
 □ <guarded cmd>
od
<stmt>
```

Finally, inserting an "IF" template at the cursor yields:

```
i:= 0;
do <guard> →  if <guard> →  <stmt>
               □ <guarded cmd>
              fi
               <stmt>
 □ <guarded cmd>
od
<stmt>
```

The main data structure of the synthesizer is declared as follows:

var p:**struct** program:stmtlist **with** cursor

The value part (<u>program</u>) of the **struct** is a tree whose nodes are described in Figure 3.3.2. Note the correspondence between the recursive data types described in Figure 3.3.2 and the grammar of Figure 3.3.1. Programs represented using these data types would look very much like conventional parse trees. For example, the program segment in Figure 3.3.3 would be represented by the tree given in Figure 3.3.4.

The current position being worked on in the program is kept in the built-in path <u>cursor</u>. An invariant maintained by the primary loop of the synthesizer is that cursor! is either a <u>stmtlist</u> or a <u>gcmdlist</u>; the current statement (or guarded command) is first(p.cursor!). The top level of the synthesizer is given in Figure 3.3.5.

```
constructor
    gcmd   (display:disp, guard:boolexpr, cmd:scons),
    scons  (first:stmt, rest:stmtlist),
    assign (target:boolexpr, value:expr),
    dostmt (display:disp, body:gcons),
    ifstmt (display:disp, body:gcons),
    gcons  (first:gcmd, rest:gcmdlist),
    skip   (),
    abort  (),
    ϵ-stmt (),
    ϵ-gcmd (),
    disp   (?, ...., ?);

union
    stmt      = skip ∪ abort ∪ dostmt ∪ ifstmt ∪ assign,
    stmtlist  = ϵ-stmt ∪ scons,
    gcmdlist  = ϵ-gcmd ∪ gcons,
    if_or_do  = ifstmt ∪ dostmt,
    cons      = scons ∪ gcons;
```

Figure 3.3.2
Recursive data types for program tree


```
/* establish the truth of "0≤k<n ∧ (∀i:0≤i<n : f_k ≥ f_i)" */
k:= 0; i:= 1;
do i ≠ n →  if f_k ≥ f_i  →  i:= i + 1
            □ f_k < f_i  →  k:= i; i:= i + 1
            fi
od
```

Figure 3.3.3
A small program segment

3.4. <u>Implementation</u>

In chapter 2 we placed five restrictions on the use of paths to facilitate efficient implementation:

(1) The value part of a **struct** variable must be an S-expression.

(2) Extensions to a built-in path that are not valid in the current value are prohibited.

(3) Only one path can be associated with a **struct** variable.

(4) Update operations cannot replace speared nodes.

(5) Path variables must be associated with (i.e. "built-into") a particular **struct** variable.

We now describe how these restrictions can be relaxed or eliminated by making minor changes to the implementation presented in section 2.4. At the end of this section we explain how paths fit in with some standard very-high-level language feature implementation techniques. An

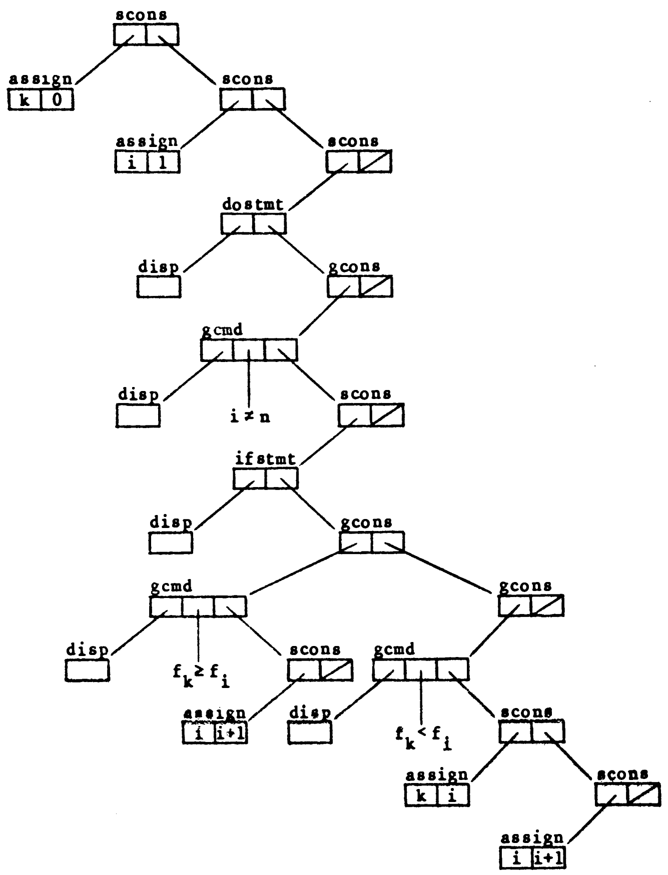


Figure 3.3.4
Representation of small program segment of Figure 3.3.3


```
nestlevel:= 0;
input(command);
do command ≠ STOP →
    if command=PREV ∧ last(cursor) = REST →
        cursor:= ↑cursor
     □ command=NEXT ∧ cursor!:cons →
        cursor:= cursor↓<REST>
     □ command=OUT ∧ nestlevel≠0 →
        do last(cursor)=CDR →  cursor:= ↑cursor od;
        cursor:= ↑↑cursor;
        nestlevel:= nestlevel-1
     □ command=IN ∧ cursor!:scons ∧ cursor↓<FIRST>!:if_or_do →
        cursor:= cursor↓<FIRST, BODY>;
        nestlevel:= nestlevel+1
     □ command=IN ∧ cursor!:gcons →
        cursor:= cursor↓<FIRST, CMD>;
        nestlevel:= nestlevel+1
     □ command=INSERT ∧ cursor!:gcons →
        cursor↓<FIRST, GUARD>!:= parse_boolean()
     □ command=INSERT ∧ cursor!:stmtlist →
        cursor!:= scons(parse_assign(),cursor!);
        cursor  := cursor↓<REST>
     □ command=INS_DO ∧ cursor!:stmtlist →
        cursor!:= scons(do_template,cursor!)
     □ command=INS_IF ∧ cursor!:stmtlist →
        cursor!:= scons(if_template, cursor!)
     □ command=INS_GC ∧ cursor!:gcmdlist →
        cursor!:= gcons(gcmd_template, cursor!)
     □ command=DELETE ∧ cursor!:cons →
        cursor!:= rest(cursor!)
     □ command=CUT ∧ cursor!:scons →
        cutbuf:= first(cursor!)
     □ command=PASTE ∧ cursor!:stmtlist →
        cursor!:= scons(cutbuf, cursor!);
        cursor := cursor↓<REST>
     □ otherwise →   beep() /* error */
    fi;
    display();
    input(command)
od
```

Figure 3.3.5
A Simple Program Synthesizer

implementation in the programming language C of the basic path operations appears in Appendix B.

The first restriction made above was really made to facilitate the <u>description</u> of the implementation rather than the implementation itself. It is no more difficult to implement paths in the context of arbitrary recursive data structures than it is to implement them in S-expressions. The only important consideration is that the paths implementation needs to know the types of nodes in the data structure. Among other things, this will allow consistency checks when a built-in path is <u>extended</u> by a particular field.

One way to allow the implementation to determine the type of a node is to partition memory into blocks, where all nodes in a block are of the same type. Thus, the implementation could infer the type of a node by its address. An alternative solution is to have a tag field in each node. While this seems conceptually simpler, it makes each node larger. (In the first scheme there is one tag per block, rather than one tag per node.)

The second restriction can be relaxed by changing the representation of a built-in path slightly. Instead of using the pair

[pointer to target, last field]

we will represent a path by the pair

[pointer to target, non-null sequence of field selectors] .

The canonical form of such a representation is

[pointer to target, <last field>] .

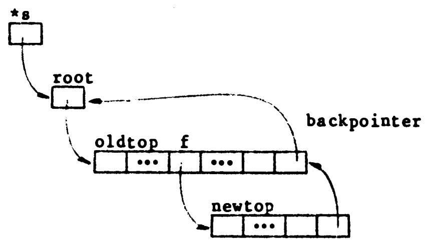
If an extension not valid in the current value is made, the field selector is appended to the sequence part of the path representation. When the path is dereferenced, the sequence will be built into the structure, restoring the path to its canonical form.

The third restriction cited above is occasionally onerous. In many cases, it is advantageous to associate more than one path with a structure. A good example is the problem of swapping two disjoint substructures within a variable. The natural, efficient way to perform this task is to embed two built-in paths within the variable. Fortunately, we can accommodate multiple built-in paths — including block-structured allocation and deallocation — by making only minor modifications to the implementation.

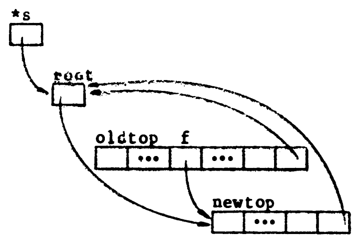
In order to maintain an accurate record of the speared status of nodes, we must keep a <u>target count</u> for each node, recording the number of built-in paths pointing to that node. Without target count information, the <u>retract</u> function cannot determine whether the target node is still speared by other built-in paths. (The target node remains speared if any of its children are speared or if any other built-in paths point to it.) Since target counts greater than 2 are unlikely to arise much in practice, a space-efficient implementation could use 2 bits to record the count and handle overflows either by storing the information in a hash table or by creating a dummy father node (accessible using the back pointer stored in the reference count field) to hold the oversize count and a back-pointer to the real father node.

The efficient implementation of operation <u>shrink</u> in the context of multiple paths requires the use of "phantom" nodes in the event that the replaced node is also a target node. The replaced target node becomes a phantom node containing a pointer to the correct path-ending node. (The "phantom" status of a node is recorded in a flag bit.) The first time a path pointing to a phantom is referenced, it is fixed up to point to the proper target node. Phantom nodes eliminate the need to search all paths associated with a structure to find the ones that need fixing up after a *shrink* or *grow* operation.

If the top part of **struct** variable *s* is as pictured below:

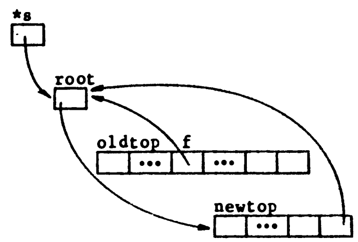


where the node labeled *root* is a dummy node used to eliminate boundary cases, then the basic *shrink* operation would yield



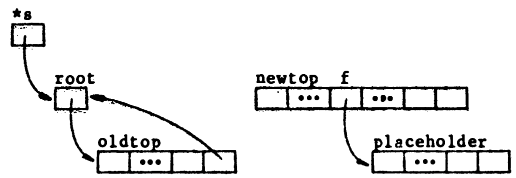
In the case where *oldtop* has a non-zero target count (i.e. there is a built-in path with value <f>) then *oldtop* becomes a phantom node. Since

the path <f> is shortened to <> by the *shrink* operation, then paths pointing to *oldtop* before shrink(s) should point to *root* afterward:

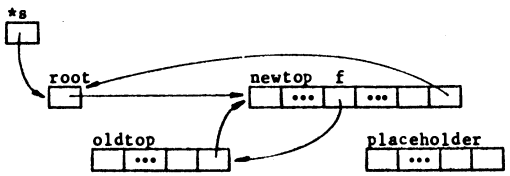


When a phantom node is used, as above, the target count of the node it points to must be increased by the target count of the phantom. This is to reflect the fact that the paths terminating at the phantom logically terminate at its target. At that point the target count of the phantom node serves as a reference count for the node. When a path target is fixed up to point to the correct target node, the target count of the phantom is decremented. When it reaches zero, the phantom node can be reclaimed.

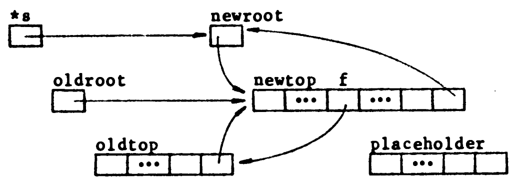
Operation *grow* employs phantom nodes as well. In the standard scenario, if *s* is a **struct** and *newtop* is a value:



then grow(s, f, newtop) would produce



where placeholder is just as its name implies; it is thrown away. If root is a target node (i.e. one or more of the built-in paths is a null path) the paths pointing to it must be fixed up to point to newtop instead; they have been extended by grow from <> to <f>. To handle this, a new root node is created and the old one becomes a phantom node:



When paths pointing to oldroot are dereferenced and fixed up, they will have their target nodes changed to be newtop. In order to be able to fix up the field part of the path value, the "f" of the grow operation must be recorded somewhere. The node oldroot seems a likely place to do this, even though it causes a slight increase in the size of all root nodes. Note that this problem does not occur with shrink. In that operation the affected paths were always changed from <f> to <>. In that case it is sufficient for the implementation to know that a phantom node points to a root node.

In chapter 2 we outlawed updates that replace speared nodes because they could make a structure inconsistent with the path (or paths if multiple built-in paths are allowed) embedded within it. To remove this restriction, we must formulate a policy on how to handle paths embedded within replaced structures. There are three basic choices:

(1) Each such path becomes undefined.

(2) Each such path is embedded in the new structure, if possible. Otherwise it becomes undefined.

(3) Each such path is embedded in the resulting structure on demand. Any attempt to dereference the path embeds it in the structure. In the meantime, extensions and retractions are permitted.

Since the semantics of paths presented in section 3.2 did not make a special case of updates at speared nodes, the third alternative is the only one that is completely consistent wtih the path semantics we have presented.

To implement to third alternative, all paths pointing to nodes in the replaced structure must be marked as being temporarily invalid when an update operation attempts to replace a speared node. Since the target nodes corresponding to invalid paths are much easier to find than the path variables themselves, we indirectly invalidate the appropriate path variables by searching the replaced structure and marking all the target nodes as "temporarily invalid". In addition, the replaced speared node is marked as being the top of a replaced structure. Its back pointer is kept intact. When the temporarily invalid path is dereferenced, the path is retracted back up to the father of the replaced node, stacking the field selectors in the process. When the replaced node is reached, the path is extended down the new value, popping the field selectors from the stack. Extensions and retractions to the path that are made while it is temporarily invalid can be handled in the usual manner.

Updates at speared nodes are clearly less efficient than updates at unspeared nodes. Instead of constant time, they take time proportional to the number of speared nodes in the replaced structure.

The fifth restriction — the one that requires paths to be built-in — is the most difficult to relax. In the context of compiling programs with paths, we conjecture that it is often feasible to optimize the implementation of a general path variable by embedding its representation in one or more value variables (just like a built-in path). Moreover, a compiler should be able to optimize the implementation of built-in paths in contexts where some operations (such as _retract_) are not needed. The global optimization of path programs is an interesting topic for future research.

Since we are proposing the use of paths in very-high-level programming it seems prudent to investigate how well they mesh with other very-high-level features. Of particular interest is the interaction between paths and space-saving implementation schemes. Perhaps the most interesting space-saving technique is hash-consing [2]. When hash-consing is used, the _cons_ function is responsible for ensuring that no list structure is duplicated. If the result of a _cons_ operation already exists, then _cons_ returns a pointer to that node rather than creating a new one.

Hash-consing is important because we envision the map data type described in section 3.1 as being implemented that way. The use of hash-consing guarantees that two equivalent values are represented by the same structure. In a hash-consing environment, LISP's _eq_ and _equal_ functions are identical. Fast equality checks are essential for the implementation of maps.

Unfortunately for paths, which require _unshared_ speared nodes, hash-consing seeks maximal sharing. To accommodate paths in a hash-consing environment, we must make modifications to our path implementation. We will hash-cons all the _shareable_ nodes of structures. Speared nodes will be represented by a larger node. Included in the speared node will be the address of its hash-consed equivalent. Thus equality checks can still be made in constant time.

Alternatively, if the space penalty of larger speared nodes is too great, the implementation can revert to their standard representation and modify the equality check. The new _eq_ would have to behave like LISP's _equal_ if one or both of its arguments was a speared node. Thus an equality check would take time proportional to the number of speared nodes in the values being compared.

3.5. _Assessment_ (_or What Price Abstraction?_)

The motivation behind paths was the desire to transform certain disciplined uses of pointers into abstract operations, without sacrificing the efficiency of the original pointer operation. We proposed implementing a path as a pointer plus a small amount of overhead. An important consideration for some potential users is the penalty imposed by the "small amount of overhead".

In a gross attempt to answer that question, a comparison was made between the running times of programs using pointers and their path counterparts. The particular programs used were the two versions of Luckham and Suzuki's event queue described in chapters 1 and 2. Both programs were written in C; the only existing implementation of paths extends C with procedures that provide path operations.

In the first comparison, the small amount of overhead expected was, in fact, quite expensive. The paths version of the event queue took nearly sixty times as long to execute as did its pointer counterpart. Since the paths implementation was very modular — many short procedures and functions were employed — we determined that much of the time was being spent doing procedure calls. A transformation of some functions into in-line code (by using a macro facility) reduced the overhead to a factor of ten, a much more tolerable figure.

All things considered, a factor of ten of overhead with the implementation used is really quite encouraging. Even at that level it is acceptable to use paths for many kinds of programming. The simple program synthesizer of section 3.3 ran with no perceptible delays. In highly interactive cases such as that, a reasonable processor is sufficiently fast to make up even the original sixty-fold penalty.

Although a ten-fold penalty may not seem "tolerable" to some people, it should be noted that the path implementation that was used was an experimental one and was not written with efficiency in mind. We

feel that a new implementation could cut overhead costs by another 50%. Moving the implementation into a compiler would reduce overhead costs still further. An intelligent compiler could, for example, identify those structures that had only one built-in path. Significant savings can be achieved by the path operations in those cases.

Further speed-up could be achieved by moving the parts of the implementation into microcode. At that level, much of the overhead of checking checking and setting bits becomes insignificant when compared to a single memory access. We estimate that microcoded path primitives would reduce the overhead to a factor of two or three. Machines with high-level, vertical microcode, such as the IBM 801 minicomputer [29], are good candidates for such an implementation.

CHAPTER 4

THEORETICAL INVESTIGATIONS

Although the recommended implementation of paths presented in Section 3.4 is a practical one, its worst-case asymptotic time complexity is not very attractive. Pathological operations, such as extracting a value at a speared node near the root, can require time proportional to the length of the path. A sequence of N such operations would require $O(N^2)$ time. In the third section of this chapter an asymptotically efficient implementation is presented. The first two sections present efficient implementations of queues and deques in Pure LISP, upon which the efficient path implementation is based.

The linear-time algorithm of section 4.1 is similar to a Turing Machine construction due to Stoss [31]. Since a one-tape TM can be simulated in real time in Pure LISP, the multi-head TM simulation results of Leong and Seiferas [24] implies the existence of a real-time queue implementation. Their work, which is much more general, leads to a queue implementation significantly more complicated than the one developed here.

4.1. A Real-Time Queue in Pure LISP

In Pure LISP, the inability to access the end of a list in constant time increases the asymptotic complexity of some algorithms. It is impossible, for example, to append one list to another one without rebuilding the first. If the end of the first list could be accessed (and modified) then the append operation would take constant time rather

than time proportional to its length. This restriction on access is of particular significance when implementing a queue. By its very nature a queue requires the ability to access both ends of a list.

Three queue operations are implemented: Query, Delete, and Insert. Query[q] yields the element at the front of queue q. Delete[q] yields q with its front element removed. Insert[q, v] yields the queue formed by adding the element v to the end of queue q. A straightforward implementation of these operations is given in Figure 4.1.1.

If the head of the queue is kept at the front of a list, as is done in our implementation, then the Delete and Query operations can be performed in constant time. The Insert operation, however, requires rebuilding the queue, and therefore takes time proportional to its length.

A fairly simple modification allows constant time access to both ends of the queue in most circumstances. A queue with value $q_1, \ldots, q_i, q_{i+1}, \ldots, q_n$ is stored as a head list $(q_1 \ldots q_i)$ and a reversed tail list $(q_n \ldots q_{i+1})$ with the head list empty only if the tail list is also.

```
Query[q]      ≡ car[q]
Delete[q]     ≡ cdr[q]
Insert[q, v]  ≡ append[q, list[v]]
```

Figure 4.1.1
Queue operations in Pure LISP

Insert can now be performed in constant time by <u>cons</u>'ing an element onto the tail list. Since the head list is empty only if the entire queue is empty, Query can be performed in constant time. Delete is performed by deleting the element q_1 and, if the head list becomes empty, reversing the tail list and making the result the head list. Figure 4.1.2 shows operations that manipulate queues represented this way.

This implementation reduces the cost of n operations from $O(n^2)$ to $O(n)$. This result follows from the fact that a list of length k is reversed only when there have been k Inserts since the last reversal operation. Since the reversal of the list of length k can be done in O(k) steps, the implementation expends O(k) time to reverse a list only after spending O(k) time to construct the list being reversed. Since the other operations (Query and Insert) require only constant time, n queue operations can be done in O(n) time.

```
Query[q] ≡ car[car[q]] --return car of H

Insert[q, v] ≡ cons[car[q], cons[v, cdr[q]]] --tack v onto T

Delete[q] ≡
   if null[cdr[car[q]]] --one left in H
      then cons[reverse[cdr[q]], NIL] --H:= reverse(T); T:= NIL
      else cons[cdr[car[q]], cdr[q]]  --H:= cdr(H);
   fi
```

A queue is (H . T) where H is the head and T the tail (reversed)

Figure 4.1.2
Linear time queue operations

The critical modification required to achieve a real time implementation[1] is distributing the reversal of the tail list over a number of operations. The basic idea of performing one step of the reverse during every operation is easy to implement, if the list being reversed is not changing. For example, if L is of length n, then

$$\underbrace{\text{incr\_rev[incr\_rev[} \ldots \text{incr\_rev}}_{\text{n times}}\text{[cons[L, NIL]]} \ldots \text{]]}$$

will return with (NIL . L_R), where L_R is the reverse of L, if incr_rev is defined as follows:

incr_rev[X] ≡ cons[cdr[car[X]], cons[car[car[X]], cdr[X]]].

Note that any one call to incr_rev is performed in constant time.

Rather than wait for the head list H to empty, we periodically Reverse the tail list T and append it to the head list, forming a new head list H'. This is a three-step process:

(1) Reverse T to form the tail end of H'.
(2) Reverse H to form H_R.
(3) Reverse H_R onto the front of H'.

Each of these operations is just the reversing of a list and can be done using a function like incr_rev. The first two steps are independent and

---

[1] A <u>real-time</u> queue implementation performs any sequence of Inserts, Deletes, and Querys with only a constant amount of processing between operations; note that this is stronger than linear time. In the case of LISP, we assume the existence of real-time car, cdr, and cons primitives [3].

can be done in parallel. Thus, if at the start of the reversal process we have

```
        front           rear
        q0 ··· qm  qm+1 ··· qn
        +-{ H }--   ---{ T̃ }--+
```

(where T̃ indicates that the depiction of T is reversed,[2]) then halfway through the above process we have:

```
        front            rear
        q0 ··· qm  qm+1 ··· qn
        --{ H̃R }--+  +--{ H' }--
```

Since q_m is at the front of H_R and q_{m+1} is at the front of H' then cons(car(H_R), H') will produce (q_m q_{m+1} ··· q_n). After m+1 cons operations we have:

```
        front            rear
        q0 ··· qm  qm+1 ··· qn
        +--------{ H' }--------
```

This is the basic idea behind implementing the real-time operations. One minor hitch remains, however! while performing the incremental reverses the queue will not remain constant. The representation needs to reflect the current state of the queue as well as the current state of the rebuilding operation.

Taking care of the Inserts that occur while rebuilding is simple. A new tail list T' is used and the new elements cons'ed onto it instead of the old tail T. In order to be able to satisfy Query and Delete

[2]Note that list H can have its leftmost elements accessed while a reversed list such as T̃ has its rightmost elements accessible. The plus signs ("+") in the list indicate which end is accessible.

requests, there must be two copies of the head list. One head list (h) represents the front end of the queue. Deletes will be performed by replacing h by cdr(h), and Query's performed by returning car(h). The other head list (H) is used in the reversal process to form H_R. Since there may have been Deletes, not all of H_R should be reversed onto H'. The length of h is kept in the variable #copy to indicate how much of H_R needs to be copied.

Along with the rebuilding process we need a strategy to indicate when rebuilding should be done. We choose to start rebuilding when the length of the tail list T is greater than the length of the head list H. Since one rebuilding process must be completed before another starts, it is necessary to show that the rebuilding operation leaves us with $|H'| \geq |T'|$. When the queue is not in "rebuild mode", it will behave like the queue of Figure 4.1.2.

Using this strategy the rebuilding operation starts when the tail list has length n+1 and the head list has length n. Simultaneously reversing H and T in the first pass of the rebuilding will take n+1 incremental steps to perform. Reversing H_R in the second pass will take n steps to perform. Thus there are a total of 2n+1 steps in the rebuilding process. The implementation will not be able to access the front of the queue in constant time if the head list h is emptied before the rebuilding is completed. Since h is n elements long, the rebuilding process must perform 2n+1 incremental steps in at most n queue operations. This will be accomplished by performing the first two steps of the rebuilding process in the operation that causes T to be longer than H, and two more steps of the process during every subsequent queue operation. This gives us $2 \times (n+1)$ incremental stepa before the head list will be emptied. This schedule is sufficient to ensure that the rebuilding process will be completed in time.

When the queue's tail list becomes one element longer than its head list, as shown below, rebuilding begins.

```
/q0                             /qn                         /q2n
+------------H-------------  ------------T------------+
```

After a_1 additions to the queue and d_1 deletions (where $a_1+d_1 < \frac{n}{2}$) the queue will be in the state shown below:

```
/q0  /qd1                     /qn                        /q2n  /q2n+a1
-------HR------+ +----H----  ----T-----+ +-----H'------ --T'-+
|+ 2 x (a1+d1) +|                        |+2 x (a1+d1)+|
-del- -------------------the queue----------------------------
      +---------h----------
```

H is being reversed forming H_R and T is being reversed forming H'. New elements are cons'ed to T' and deleted elements are removed from h. Also taking place during this pass is the counting of the length of H' (in lendiff) as it is being formed, and the length of H_R (in #copy) as well.

When $a_1+d_1=\frac{n}{2}$ the queue will have finished the first pass of the rebuild operation and will be as pictured below:

```
/q0  /qd1                     /qn                        /q2n  /q2n+a1
------------HR-----------+ +-----------H'------------ --T'-+
-del- -------------------the queue----------------------------
      +---------h----------                     H = T = ()
```


After $a_2 + d_2$ more additions and deletions, the queue is in the midst of the second pass of rebuilding and is in a state as shown below:

```
/q0         /qd1+d2          /qn-1                      /q2n     /q2n+a1+a2
-------HR------+ +----------------H'------------------ ---T'--+
        +|#copy|+
-deletes- ----------------------the queue----------------------
          +-------h--------                       H = T = ()
```

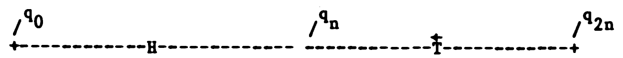
Stored in #copy is a count of the number of elements of H_R that have not been deleted from the queue. H_R is being reversed onto H' until #copy = 0. At that time H' and T' can replace H and T as the representation of the queue, as shown shown below:

```
/q0         /qd1+d2          /qn-1                      /q2n     /q2n+a1+a2
----HR--+ +---------------------H'--------------------- ---T'--+
-deletes- ----------------------the queue----------------------
          +-------h--------                       H = T = ()
```

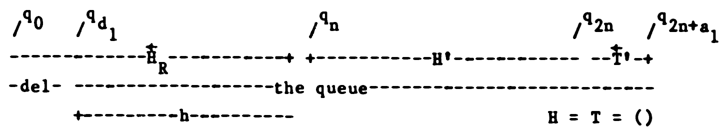
The actions of the rebuilding process are summarized in Figure 4.1.3. Since the new head list is of length $2n+1-d_1-d_2$ and the new tail list is of length a_1+a_2 and $d_1+d_2+a_1+a_2 = n$ then:

$$2n+1 \geq n$$
$$2n+1 \geq a_1+a_2+d_1+d_2$$
$$2n+1-d_1-d_2 \geq a_1+a_2$$

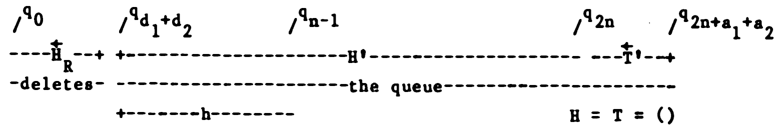
Thus, $|H'| \geq |T'|$ which we needed to show to prove that our rebuilding strategy was a valid one. An actual Pure LISP program implementing a real time queue appears in Appendix C.



(a) Before rebuilding begins



(b) After the first pass of rebuilding



(c) After rebuilding is finished

Figure 4.1.3
The queue at various stages of the rebuilding process

4.2. A Real Time Double-Ended Queue in Pure LISP

The symmetric operations of a double-ended queue (deque) require a different implementation strategy from the one presented for the queue. The approach of Section 4.1 relied on the monotonicity of queue operations; Insert and Delete lengthen the tail list relative to the head. This fact led to an implementation where it was sufficient to keep the head list longer than the tail. Since the two ends of a deque are symmetric, it seems reasonable to expect that an asymptotically efficient deque implementation will be symmetric as well.

The linear time deque implementation is nearly identical to its queue counterpart. The deque is stored as a head list and a reversed tail list. A rebuilding operation is initiated when either the head list or the tail list reaches length 0. Rebuilding generates new head and tail lists of (approximately) the same length.

To make the linear time implementation into a real time one, we will again use the strategy of distributing the rebuilding costs. A rebuilding operation will commence when the two lists are deemed "unbalanced". Upon its completion the lists will be "balanced". If H is the shorter of the two lists, and the combined lengths of H and T is N, rebuilding (ignoring intervening deque operations) proceeds as follows:

1) Reverse H forming H_R; reverse first $\frac{N}{2}$ of T forming T_R and T
2) Reverse rest of T forming H'; start reversing T_R forming T'
3) Reverse H_R onto H'; continue reversing T_R onto T'

The above static rebuilding process is pictured in Figure 4.2.1.

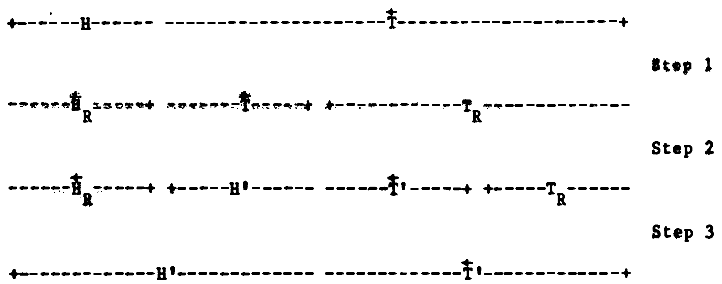


Figure 4.2.1
Static rebalancing of a double-ended queue

As with the queue, we must make special arrangements to accommodate changes to the deque that occur while rebuilding is underway. We will use a stack at each end to keep the true state of the deque during rebuilding. Additions are performed by pushing onto the stack. If the stack is non-empty, deletions are reflected by popping from the stack. Deletions with an empty stack are handled as they were with the queue: a count is kept of their number, and copies of H and T reflecting their true states are maintained for purposes of implementing Query correctly. The final forming of H' and T' throws out the deleted elements.

If there is a non-empty stack after step 3 of the rebuilding process, then the stack must be appended to the appropriate list before rebuilding is completed. Unfortunately, while the appending (2 incremental reverses) is taking place, other deque operations may be making changes. Again a stack of net additions will be kept. Thus the rebuilding process looks like this:

1) Reverse H forming H_R; reverse first $\frac{N}{2}$ of T forming T_R and T
2) Reverse rest of T forming H'; start reversing T_R forming T'
3) Reverse H_R onto H'; continue reversing T_R onto T'
   (new net additions go into stacks H_4 and T_4)
4) "Fixup" H' and T' by appending temporary stacks H_4 and T_4
   (new net additions go into stacks H_5 and T_5)
5) Fixup H' and T' by appending temporary stacks H_5 and T_5
   (new net additions go into stacks H_6 and T_6)

⋮

k) Fixup H' and T' by appending temporary stacks H_k and T_k
   (new net additions go into stacks H_{k+1} and T_{k+1})

⋮

If appending the first stack takes place fast enough, we can guarantee that the second stack formed is smaller than the first. Similarly the third stack will be smaller than the second, and so on.

The only remaining question in the deque implementation is when and how fast rebuilding must take place. We choose to begin rebuilding when one list is three times the length of the other, and we will perform 4 rebuilding steps per deque operation. If the deque is of length N, step 1 of rebuilding takes $\frac{N}{2}$ incremental steps (which can be performed in $\frac{N}{8}$ deque operations since we are performing 4 rebuilding steps per deque operation.) Similarly, steps 2 and 3 take a total of $\frac{N}{2}$ steps ($\frac{N}{8}$ deque operations). Note that the choice of parameters guarantees that steps 1-3 of rebuilding can be completed (they require $\frac{N}{4}$ deque operations) before the shorter of the two lists (of size $\frac{N}{4}$) could be overdrawn by deletions. Since steps 1-3 are performed in $\frac{N}{4}$ operations, the larger of H_4 and T_4 can be at most of size $\frac{N}{4}$. Thus step 4 requires $\frac{N}{4}$ steps ($\frac{N}{16}$ deque operations) and produces stacks H_5 and T_5 of size $\le \frac{N}{16}$. For $k \ge 4$, step k requires $\frac{J}{4}$ deque operations, where J is the size of the larger of H_k and T_k. Thus the total number of deque operations required for steps 4 and up is

$$\frac{N}{16} + \frac{N}{64} + \frac{N}{256} + \dots = N \times \sum_{i=2}^{\infty} 4^{-i} = \frac{N}{12}.$$

It remains to show that when the "fixup" operation of appending the stacks to the appropriate lists is completed, the deque is not "unbalanced".

If the deque is of length N when rebuilding starts, (a list of size $\frac{N}{4}$ and one of size $\frac{3N}{4}$) we have seen that rebuilding takes **at most** $\frac{N}{3}$

($\frac{N}{4}$ for steps 1-3 + $\frac{N}{12}$ for steps 4 and up) deque operations to complete. We will perform the first rebuilding steps during the operation that causes the deque to be unbalanced. Thus there are $\frac{N}{3}$ - 1 deque operations during rebuilding. The worst operation for maintaining balance of the two lists is Delete. (since if $n \geq m > 0$ then $\frac{n}{m-1} > \frac{n+1}{m}$.)[3] If all $\frac{N}{3}$ - 1 operations were deletions from one side[4], say the tail, then H would be of size $\frac{N}{2}$ at the completion of rebuilding. The tail list T would be of size $\frac{N}{2} - (\frac{N}{3} - 1) = \frac{N}{6} + 1$. Since the head list is less than 3 times the size of the tail, the deque is "balanced".

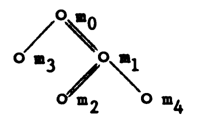
4.3. An Alternative Implementation of Paths

Instead of using explicit pointers and records, as was done in sections 2.4 and 3.4, the approach of this section is to represent paths as lists in Pure LISP. An S-expression with a single embedded path is very much like a deque — primitive operations must be able to access values near each end of the path. The values on the path near the root of the structure are used by cons, car, cdr, shrink, and grow. The values near the other end are used by extract, update, extend, and retract. Therefore, like the queue and deque, the representation of the S-expression part is broken into two pieces. The part of the S-expression containing the top half of the path (kept in the structure TOP) is modified slightly. Cons nodes along the path are expanded to include information about the path. They are also arranged so that the cdr of one path node

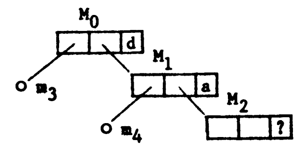
[3] This can be easily seen by cross-multiplying.

[4] This is clearly impossible, since if all the operations are Deletes, there can be only N/4 of them. It is Insert that yields 25% interest by creating temporary stacks that must be copied.

is the next node down the path. Thus if m_1 is a node on the top half of the path:

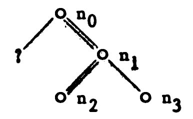


it is represented in TOP by node M_1 as follows:

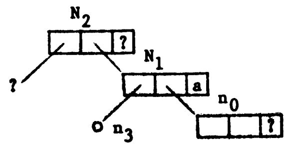


where the "a" inside M_1 indicates that the path goes down the "car" side of m_1. (Thus, an "a" inside M_i indicates that its sons are reversed relative to those of m_i.)

The part containing the bottom half of the path (kept in the structure BOTTOM) is turned upside-down — a value on the path near its end is more readily accessible than value further up the path. Inverting the bottom part of the tree turns the path node n_1:



into N_1 as follows:



As with the representation of TOP, the "a" inside N_1 indicates that the path goes down the "car" side of n_1. Figures 4.3.1 through 4.3.3 show an example structure with an embedded path and its proposed TOP and BOTTOM representations.

Since the value of the dereferenced path is near the root of BOTTOM, extend and retract can be performed efficiently. Extend merely creates a new root node whose car is the new value of the dereferenced path, and whose cdr is a new path node. The new path node's car is the brother of the dereferenced new path, and its cdr is the cdr of the old root node. The third field of the new path node is the field-selector by which the path was extended. Figure 4.3.4 shows what happens to the BOTTOM of Figure 4.3.3 when the path is extended by a "car".

The retract operation can be performed in a similar manner. The cdr of the new root node is the cddr of the old root node. The car of the new root is the cons of the car of the old root and the cadr of the old root. The ordering of the node is determined by the third field of the deleted path node. Figure 4.3.5 illustrates a retract operation performed on the BOTTOM of Figure 4.3.3. Shrink, grow, extract, and update also have straightforward implementations. A first attempt at efficient path operations is given in Figure 4.3.6. The operations

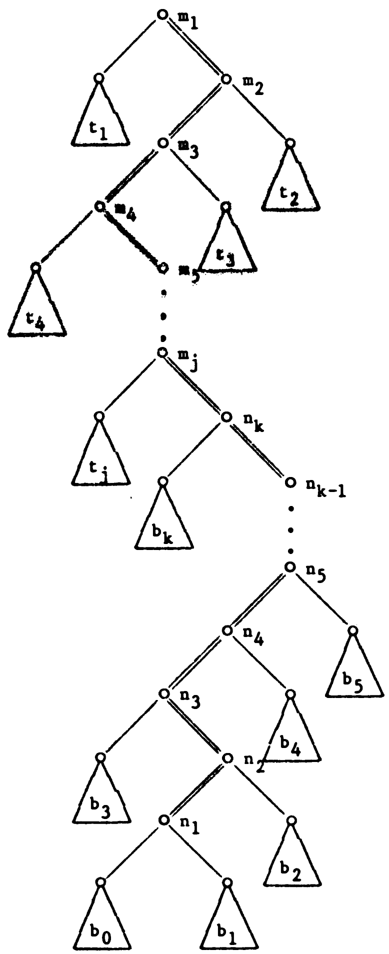


Figure 4.3.1
The structure represented in Figures 4.3.2 and 4.3.3

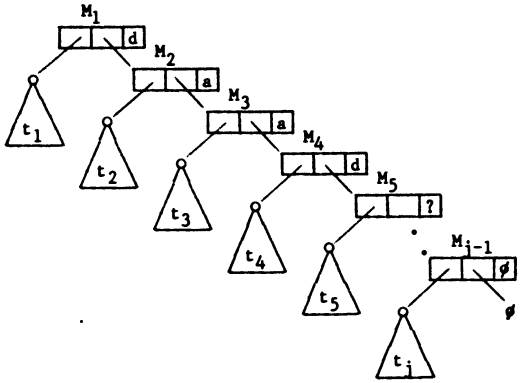


Figure 4.3.2
Proposed representation of TOP

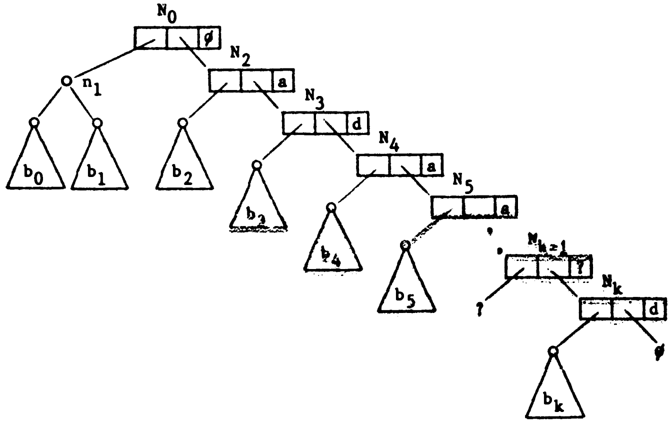


Figure 4.3.3
Proposed representation of BOTTOM

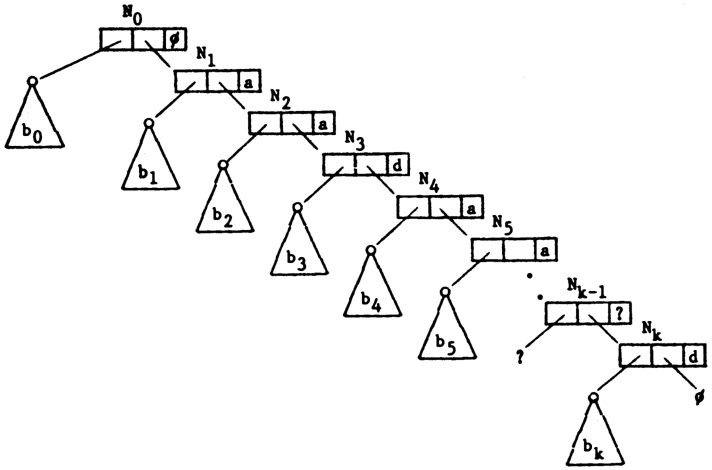


Figure 4.3.4
BOTTOM after extending path by CAR

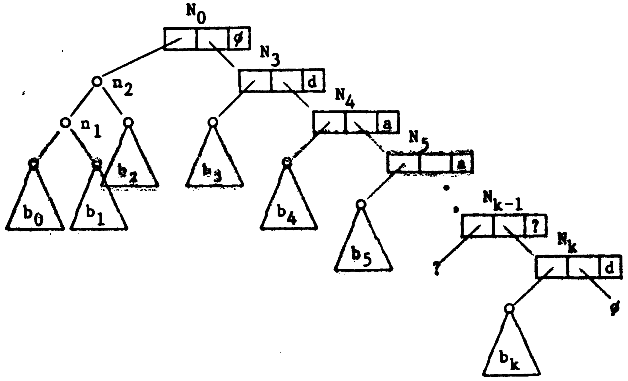


Figure 4.3.5
BOTTOM after a <u>retract</u>

```
extract(S)    ≡car[BOTTOM]
update(S)     ≡cons[TOP, cons[x, cdr[BOTTOM]]]
shrink(S)     ≡cons[cdr[TOP], BOTTOM]
grow(S, f, x) ≡cons[pathnode[ f[x], TOP, f], BOTTOM]
extend(S, f)  ≡cons[TOP,
                  pathnode[ f[car[BOTTOM]],
                            pathnode[! f[car[BOTTOM]], cdr[BOTTOM], f],
                            ∅]]
retract(S)    ≡cons[TOP,
                  pathnode[ fcons[ field[cdr[BOTTOM]],
                                   car[BOTTOM], cadr[BOTTOM]]],
                           cdr(BOTTOM),∅]]
```

Figure 4.3.6
Moderately efficient path operations

there assume that a **struct** S is a cons(TOP, BOTTOM). Note that each operation takes time independent of the length of the path.

With this representation of **struct** variables, cons, car, and cdr must be slightly modified. When part of a **struct** is needed as a pure value, a function that translates a **struct** into an S-expression must be called.

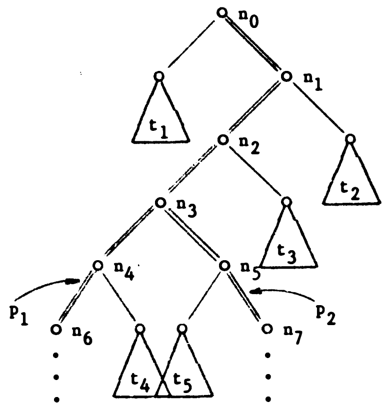
The correctness of the path operations of Figure 4.3.6 depends on not hitting a boundary case. For example, if a retract operation is requested and there is no bottom-part path, then the implementation in Figure 4.3.6 would fail. Retract (and the other operations) could be patched so that they rebuilt the structure whenever an operation failed

because the structure was out of balance. Although rebalancing takes time proportional to the length of the path, after rebalancing a path of length N, at least $\frac{N}{2}$ operations would be required to force another rebuilding. Thus even though one of N path operations might take time O(N), the total time required for all N operations is still O(N) in the worst case.

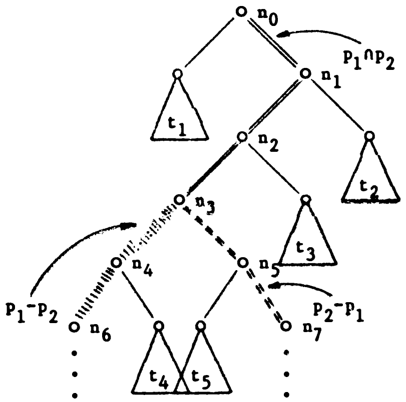
The expensive path operations that force rebuilding the structure can be avoided if the structure is kept balanced incrementally, as was done with the real time deque. When the structure becomes unbalanced, the building of a new structure can be started. Progress in its construction will be made incrementally, adding only constant cost to each path operation performed. By the time it would be needed the new, balanced structure will be completed.

The modified cons, car, and cdr suggested earlier must be modified so they can perform translations from **struct**'s to S-expressions incrementally. Such an incremental process would have time proportional to the length of the path in TOP in which to turn BOTTOM right-side-up.

Modifications to the foregoing implementation allow the representation of S-expressions with multiple embedded paths. In the skeletal tree formed by the paths in a structure, we need to isolate different segments and implement each as a deque. For example, if a structure has 2 embedded paths p_1 and p_2:



the segments of interest are $p_1 \cap p_2$, $p_1 - p_2$, and $p_2 - p_1$:



If there are more than two embedded paths in a structure, the technique is similar — the skeletal path tree is partitioned into the

segments found between the nodes of the structure where the path tree branches. In general, the number of segments to maintain of grows exponentially with the number of embedded paths in a structure. At first glance this appears to destroy our real time simulation. The problem is inconsequential, however. The length of the program increases exponentially, but each path operation still requires only constant time. (The constant, although perhaps large, is independent of the input to the path program.)

The implementation of paths suggested in this chapter was, at first glance, made more difficult than necessary. The choice of Pure LISP as the underlying machine (rather than a RAM) made some aspects of the implementation much harder. Unlike a RAM implementation, however, using a Pure LISP machine allows us to make the following statement: "Any programs written in LISP+Paths can be translated into Pure LISP so that it runs with only constant degradation of speed." That is, from a purely theoretical standpoint, while addition of paths to LISP may make the programming easier, it does not increase the "power" of the resulting language.

From a practical standpoint, we could never seriously recommend using the implementation of this chapter. The constant of "constant degradation" is rather large. The implementation of section 3.4, although not real time, is much better.

CHAPTER 5

SUMMARY AND DIRECTIONS FOR FURTHER RESEARCH

5.1. Summary of Results

The main focus of the research was the investigation of efficiently-implementable very-high-level programming language constructs. In a manner analogous to ALGOL 60's abstraction away from goto's, we proposed that the path be used as an abstract replacement for a typical use of pointers in high-level languages. The use of paths instead of pointers in unshared recursive data structures greatly simplifies the process of reasoning about programs. The existence of an efficient implementation of paths makes their use palatable as well as desirable. Because efficiency of very-high-level languages is an important consideration, much of the research effort went into the efficient implementation of paths, especially in the context of compatibility with other desirable very-high-level constructs.

One of the motivations of the research was the hope that the introduction of paths to LISP would make more efficient algorithms possible. It was suspected, for example, that a queue implementation using paths would be asymptotically more efficient than one without. The discovery of a real-time queue implementation in Pure LISP overturned that conjecture. It also led to a much more surprising theoretical result: any program written in LISP that used paths can be translated into Pure LISP so that the new program simulates the old one in real time. In other words, from a theoretical standpoint, the addition of paths to LISP does not produce a more powerful language.

5.2. Directions for Further Research

Future research in very-high-level language design could continue along either of two paths:

(1) Improve the current paths implementation by investigating the implementation of unbound paths. This would likely involve the use of global data-flow optimization techniques.

(2) Isolate other disciplined uses of pointers and provide abstract, but efficiently-implementable, mechanisms for replacing them.

A cursory investigation indicates that the first alternative could be fruitful. We hope to continue research along those lines with some optimization experts.

The continuation of the theoretical research presented in chapter 4 should include an investigation of the $\leq_{rt}$ reduction. (We will say "machine-model A $\leq_{rt}$ machine-model B" if and only if programs running on machine-model A can be simulated in real time on machine-model B.) Ultimately it is hoped that a theorem of the following form could be proved:

single-tape TM's $<_{rt}$ LISP $=_{rt}$ LISP+Paths $<_{rt}$ RAM's.[1]

The difficulty of such a proof would be showing the existence of problems whose lower bounds on one machine exceeded the complexity of an existing algorithm on the more complicated machine.

[1] Note that "LISP $=_{rt}$ LISP+Paths" was shown in section 4.3.

REFERENCES

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Allen, J. The Anatomy of LISP. McGraw-Hill, New York, 1978.

[3] Baker, H. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.

[4] Boyer, R. and Moore, J. Proving Theorems about LISP Functions. J. ACM 22, 1, 122-144.

[5] Cartwright, R. A Constructive Alternative to Axiomatic Data Type Definitions. 1980 LISP Conference Proceedings, August 1980, Stanford University.

[6] Cartwright, R. User-Defined Data Types as Aid to Verifying LISP Programs, in Automata, Languages, and Programming, S. Michaelson and R. Milner, Eds. Edinburgh Press, Edinburgh.

[7] Cartwright, R. A Practical Formal Semantic Definition and Verification System for Typed LISP. AI Memo 77-296, Stanford U., 1976.

[8] Cartwright, R. and Hood, R. The Path: An Efficient, Very High Level Construct to Replace Pointers. Submitted to TOPLAS in August 1981.

[9] Cartwright, R., Hood, R., and Matthews, P. Paths: An Abstract Alternative to Pointers. Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages. Williamsburg, January, 1981.

[10] Cartwright R. and McCarthy, J. First Order Programming Logic. POPL Conference Proceedings, January 1979.

[11] Cartwright, R. and Oppen, D. The Logic of Aliasing. Acta Informatica 15, 365-384.

[12] Dijkstra, E. A Discipline of Programming. Prentice-Hall, New York, 1976.

[13] Floyd, R.W. Assigning Meanings to Programs. Proceedings of the Symposium on Applied Mathematics, 19, J.T. Schwartz (ed.). American Mathematical Society, Providence, RI, 19-32.

[14] Gries, D. and Levin, G. Assignment and Procedure Call Proof Rules, Trans. on Prog. Lang. and Sys. 2, 4 (October 1980), 564-579.

[15] Hoare, C.A.R. An Axiomatic Approach to Computer Programming. Comm. ACM 12, 10 (Oct. 1969), 332-329.

[16] Hoare, C.A.R. Proofs of Correctness of Data Representation. Acta Informatica 1, 271-281

[17] Hoare, C.A.R. Recursive Data Structures. CS Report Stan-CS-73-400, Stanford U., 1973.

[18] Hood, R. and Melville, R. Real Time Queue Operations in Pure LISP. Information Processing Letters 13, 2 (November 1981), 50-54.

[19] Jensen, K. and Wirth, N. PASCAL: User Manual and Report. Springer-Verlag, New York, 1974.

[20] Kieburtz, Richard B. Programming Without Pointer Variables. Proceedings of the Conference on Data: Abstraction, Definition and Structure, Salt Lake City, March 1976.

[21] Kennedy, K. and Schwartz, J. An Introduction to the Set Theoretical Language SETL. Comp. and Maths. with Appls. 1, 1, 97-119.

[22] Kernighan, B. and Ritchie, D. The C Programming Language. Prentice-Hall, Englewood Cliffs, 1978.

[23] Knuth, D. The Art of Computer Programming, Vol. I. Addison-Wesley, 1973.

[24] Leong, B. and Seiferas, J. New Real-time Simulations of Multi-head Tape Units. Proceedings of the Ninth Annual Symposium on Theory of Computing, Boulder, 1977, 239-248.

[25] Luckham, D. and Suzuki, N. Verification of Array, Record, and Pointer Operations in Pascal. Trans. on Prog. Lang. and Sys. 1, 2 (October 1979), 226-244.

[26] McCarthy, J., et al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1965.

[27] McCarthy, J. A Basis for a Mathematical Theory of Computation, in Computer Programming and Formal Systems, P. Brafford and D. Hirschberg Eds. North-Holland, Amsterdam, 1963, pp. 33-70.

[28] Naur, P., Ed. Revised Report on the Algorithmic Language ALGOL 60. Comm. ACM 6, 1 (Jan. 1963), 1-17.

[29] Radin, G. The 801 Minicomputer. Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, March 1982.

[30] Schwartz, J. On Programming (Installment I: Generalities). Interim Report on SETL, New York University., 1973.

[31] Stoss, H-J. K-band-Simulation von k-Kopf-Turing-Maschinen. Computing 6, 309-317 (1970).

[32] Teitelbaum, R. and Reps, T. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Comm. ACM 24, 9 (September 1981), 563-573.

APPENDIX A

PROOF OF PATHS VERSION OF EVENT QUEUE

In this appendix we present a proof that the program in figure 2.3.2 actually implements an event queue. Specifically, we must show

```
pre ⇒ wp("search(q, x)", post)                    (A.1)
```

where **pre** and **post** are defined by:

```
pre : is_event_queue(q) ∧ q0 = q ∧ is_int(x)
post: is_event_queue(q) ∧ count(q, x) = count(q0, x) + 1
                        ∧ ∀y≠x(count(q, y) = count(q0, y))
```

and is_event_queue and count are defined by:

```
is_event_queue(q) ≡
    if q = NIL then true
     else is_dotted_pair(car(q)) ∧ is_event_queue(cdr(q))

count(q, x) ≡
    if q = NIL then 0
    else if caar(q) = x then cdar(q)
    else count(cdr(q), x)
```

Several additional functions are necessary:

```
is_not_in(x, q, p) ≡
    if p = <> then true
     else if x = q[(↑p)+<CAR, CAR>] then false
     else is_not_in(x, q, ↑p)

is_dotted_pair(s) ≡
    not atom(s) ∧ is_int(car(s)) ∧ is_int(cdr(s))

is_all_cdrs(p) ≡
    if p = <> then true
     else head(p) = CDR ∧ is_all_cdrs(tail(p))
```


Figure A.2 shows the program of figure 2.3.2 annotated with intermediate assertions. To show (A.1) it is sufficient to show (for $i = 1, \ldots, 5$):

$$\mathrm{pre}(S_i) \Rightarrow \mathrm{wp}(S_i, \mathrm{post}(S_i))$$

for each statement S_i, where $\mathrm{pre}(S_i)$ is the assertion immediately preceding S_i and $\mathrm{post}(S_i)$ is the one immediately following.

```
     {is_event_queue(q) ∧ q0 = q ∧ is_int(x)}
(1)  p := <>;
     {is_event_queue(q) ∧ q0 = q ∧ count(q, x) = count(q[p], x)
              ∧ is_valid(p, q) ∧ is_all_cdrs(p) ∧ is_int(x)
              ∧ is_not_in(x, q, p)}
(2)  do q[p] ≠ NIL ∧ q[p+<CAR, CAR>] ≠ x →  p:= p+<CDR> od;
     {is_event_queue(q) ∧ q0 = q ∧ count(q, x) = count(q[p], x)
                      ∧ is_valid(p, q) ∧ is_all_cdrs(p) ∧ is_int(x)
                      ∧ is_not_in(x, q, p)
                      ∧ (q[p] = NIL ∨ q[p+<CAR, CAR>] = x)}
     if q[p] = NIL →  t:= cons(x, 0)
(3)   □ q[p] ≠ NIL →  t:= q[p+<CAR>];q[p]:= q[p+<CDR>]
     fi;
     {is_event_queue(q) ∧ count(q0, x) = cdr(t) ∧ x = car(t)
                      ∧ ∀y≠x(count(q, y) = count(q0, y))
                      ∧ ¬is_atom(t) ∧ is_int(car(t))
                      ∧ is_int(cdr(t))}
(4)  q:= cons(t, q);
     {is_event_queue(q) ∧ q[<CAR, CAR>] = x
                      ∧ ∀y(count(q, y) = count(q0, y))}
(5)  q[<CAR, CDR>]:= q[<CAR, CDR>] + 1;
     {is_event_queue(q) ∧ ∀y≠x(count(q, y) = count(q0, y))
                      ∧ count(q, x) = count(q0, x) + 1}
```

Figure A.2
The annotated event queue program

<u>Proof of (5)</u>:

```
show:
   is_event_queue(q) ∧ q[<CAR, CAR>] = x
                     ∧ ∀y(count(q, y) = count(q_0, y))
     ⇒                                                        (A.3)
   wp("q[<CAR, CDR>]:= q[<CAR, CDR>] + 1;",
       is_event_queue(q) ∧ ∀y≠x(count(q, y) = count(q_0, y))
                         ∧ count(q, x) = count(q_0, x) + 1)
```

let q' ≡ update(q, <CAR, CDR>, q[<CAR, CDR>]+1)

Note that by the definition of update:

```
   cdr(q') = cdr(q)  ∧  caar(q') = caar(q)  ∧  cdar(q') = cdar(q)+1
```

The right-hand-side of implication (A.3) evaluates to:

```
      is_event_queue(q')
      ∧ ∀y≠x(count(q', y) = count(q_0, y))
      ∧ count(q', x) = count(q_0, x) + 1
which reduces to:
      is_dotted_pair(car(q')) ∧ is_event_queue(cdr(q'))
      ∧ ∀y≠x(count(cdr(q'), y) = count(q_0, y))
      ∧ cdar(q') = count(q_0, x) + 1
which reduces to:
      is_dotted_pair(update(car(q), <CDR>, car(q)[<CDR>] + 1))
      ∧ is_event_queue(cdr(q))
      ∧ ∀y≠x(count(cdr(q), y) = count(q_0, y))
      ∧ cdar(q) = count(q_0, x) + 1
which reduces to:
      is_dotted_pair(cons(caar(q), cdar(q) + 1))
      ∧ true
      ∧ true
      ∧ true
which reduces to:
      true
```

End of Proof of (5).

<u>Proof of (4)</u>.

```
show:
 is_event_queue(q) ∧ count(q_0, x) = cdr(t) ∧ x = car(t)
                   ∧ ∀y≠x(count(q, y) = count(q_0, y))
                   ∧ ¬ is_atom(t) ∧ is_int(car(t)) ∧ is_int(cdr(t))}
    ⇒
 wp("q:= cons(t, q);", is_event_queue(q) ∧ q[<CAR, CAR>] = x
                       ∧ ∀y(count(q, y) = count(q_0, y))
```

The right-hand-side reduces to:

```
 is_event_queue(cons(t, q) ∧ cons(t, q)[<CAR, CAR>] = x
                  ∧ ∀y(count(cons(t, q), y) = count(q_0, y))
which reduces to:
 ¬ is_atom(t) ∧ is_int(car(t)) ∧ is_int(cdr(t)) ∧ is_event_queue(q)
 ∧ car(t) = x
 ∧ ∀y≠caar(cons(t, q))(count(q, y) = count(q_0, y))
 ∧ ∀y=caar(cons(t, q))(cdr(t) = count(q_0, y))
which reduces to:
 true
 ∧ true
 ∧ ∀y≠x(count(q, y) = count(q_0, y))
 ∧ cdr(t) = count(q_0, x)
which reduces to:
 true
```

End of Proof of (4).

<u>Proof of (3)</u>:

```
show:  pre(S_3) ⇒ wp(S_3, post(S_3))

it is sufficient to show:
   pre(S_3) ⇒ BB
   ∧ (pre(S_3) ∧ q[p] = NIL ⇒ wp("t:= cons(x, 0)", post(S_3)))
   ∧ (pre(S_3) ∧ q[p] ≠ NIL ⇒ wp("t:= q[p+<CAR>]; q[p]:= q[p+<CDR>]",
                                   post(S_3)))

Part I:
   show: pre(S_3) ⇒ BB (≡ q[p] = NIL ∨ q[p] ≠ NIL)
              ⇔ true
```

Part II:

show:

(pre(S_3) ∧ q[p] = NIL ⇒ wp("t:= cons(x, 0)", post(S_3)))

the right hand side evaluates to:

is_event_queue(q) ∧ count(q_0, x) = cdr(cons(x, 0))
∧ x = car(cons(x, 0)) ∧ ∀y≠x(count(q, y) = count(q_0, y))
∧ ¬ is_atom(cons(x, 0)) ∧ is_int(car(cons(x, 0)))
∧ is_int(cdr(cons(x, 0)))

which reduces to:

is_event_queue(q) ∧ count(q_0, x) = 0
∧ x = x ∧ ∀y≠x(count(q, y) = count(q_0, y))
∧ true ∧ is_int(x) ∧ is_int(0)

which reduces to:

is_event_queue(q) ∧ count(q_0, x) = 0
∧ ∀y≠x(count(q, y) = count(q_0, y))
∧ is_int(x)

substituting q for q_0 (since they are equal by the hypothesis) yields:

is_event_queue(q) ∧ count(q, x) = 0
∧ ∀y≠x(count(q, y) = count(q, y))
∧ is_int(x)

which reduces to:

is_event_queue(q) ∧ count(q, x) = 0
∧ true ∧ is_int(x)

substituting count(q[p], x) for count(q, x) yields:

is_event_queue(q) ∧ count(q[p], x) = 0 ∧ is_int(x)

substituting NIL for q[p] yields:

is_event_queue(q) ∧ count(NIL, x) = 0 ∧ is_int(x)

which reduces to:

is_event_queue(q) ∧ is_int(x)

which is clearly implied by the hypotheses.

Part III:

show:

(pre(S_3) ∧ q[p] ≠ NIL ⇒
wp("t:= q[p↓<CAR>]; q[p]:= q[p↓<CDR>]", post(S_3))

the right hand side evaluates to:

wp("t:= q[p↓<CAR>]", wp("q[p]:= q[p↓<CDR>]", post(S_3)))

which reduces to:

wp("t:= q[p↓<CAR>]", is_valid(p, q) ∧ is_event_queue(q')
∧ count(q_0, x) = cdr(t) ∧ x = car(t)
∧ ∀y≠x(count(q', y) = count(q_0, y))
∧ ¬ is_atom(t) ∧ is_int(car(t))
∧ is_int(cdr(t))

where q' ≡ update(q, p, q[p↓<CDR>]).

The new right hand side evaluates to:

is_valid(p, q) ∧ is_event_queue(q')
∧ count(q_0, x) = cdr(q[p↓<CAR>]) ∧ x = car(q[p↓<CAR>])
∧ ∀y≠x(count(q', y) = count(q_0, y))
∧ ¬ is_atom(q[p↓<CAR>]) ∧ is_int(car(q[p↓<CAR>]))
∧ is_int(cdr(q[p↓<CAR>]))

It is thus necessary to show:

is_event_queue(q) ∧ q_0 = q ∧ count(q, x) = count(q[p], x)
∧ q[p↓<CAR, CAR>] = x ∧ is_int(x)
∧ is_valid(p, q) ∧ is_all_cdrs(p) ∧ q[p] ≠ NIL

⇒

is_valid(p, q) ∧ is_event_queue(q')
∧ count(q_0, x) = cdr(q[p↓<CAR>]) ∧ x = car(q[p↓<CAR>])
∧ ∀y≠x(count(q', y) = count(q_0, y))
∧ ¬ is_atom(q[p↓<CAR>]) ∧ is_int(car(q[p↓<CAR>]))
∧ is_int(cdr(q[p↓<CAR>]))

The proof will proceed by showing each of the conjuncts of the right-hand-side:

is_valid(p, q)
(by hypothesis)
∧ is_event_queue(q')
(by Lemma 1)
∧ count(q_0, x) = cdr(q[p↓<CAR>])
(by q_0 = q and q[p↓<CAR, CAR>] = x)
∧ x = car(q[p↓<CAR>])
(by q[p↓<CAR, CAR>] = x)
∧ ∀y≠x(count(q', y) = count(q_0, y))
(by q_0 = q and Lemma 2)
∧ ¬ is_atom(q[p↓<CAR>])
(by is_event_queue(q) and q[p] ≠ NIL and is_valid(q, p))
∧ is_int(car(q[p↓<CAR>]))
(by is_event_queue(q) and q[p] ≠ NIL and is_valid(q, p))
∧ is_int(cdr(q[p↓<CAR>]))
(by is_event_queue(q) and q[p] ≠ NIL and is_valid(q, p))

```
Lemma 1:
  ∀ q:lists, p:paths
    is_event_queue(q) ∧ is_valid(p, q)
      ∧ q[p] ≠ NIL ∧ is_all_cdrs(p)
    ⇒
      is_event_queue(update(q, p, q[p+<CDR>]))

  The proof proceeds by induction on the length of q.

  Base case: q = ()
      True because hypothesis q[p] ≠ NIL is not satisfied.

  Induction step:
    assume:
      ∀ q:lists, p:paths
        is_event_queue(q) ∧ is_valid(p, q)
          ∧ q[p] ≠ NIL ∧ is_all_cdrs(p)
        ⇒
          is_event_queue(update(q, p, q[p+<CDR>]))
    show:
      ∀ q:lists, p:paths
        is_event_queue(cons(cons(y, z), q))
          ∧ is_valid(p, cons(cons(y, z), q))
          ∧ cons(cons(y, z), q)[p] ≠ NIL ∧ is_all_cdrs(p)
        ⇒
          is_event_queue(update(cons(cons(y, z), q), p,
                                cons(cons(y, z), q)[p+<CDR>]))

      The right-hand-side reduces to:
        is_dotted_pair(car(update(cons(cons(y, z), q), p,
                                  cons(cons(y, z), q)[p+<CDR>])))
          ∧ is_event_queue(cdr(update(cons(cons(y, z), q), p,
                                      cons(cons(y, z), q)[p+<CDR>])))
      If p = <> then the right-hand-side reduces to:
        is_dotted_pair(cons(y, z))
          ∧ is_event_queue(q[<>+<CDR>])
      which reduces to:
        true
          ∧ is_event_queue(cdr(q))
      which reduces to:
        true
```



```
      If p ≠ <> then the right-hand-side reduces to:
        is_dotted_pair(cons(y, z))
          ∧ is_event_queue(update(q, tail(p),
                                  cons(cons(y, z), q)[p+<CDR>]))
      which reduces to:
        true
          ∧ is_event_queue(update(q, tail(p),
                                  q[tail(p)+<CDR>]))
      which is true by the induction hypothesis.
End of Proof of Lemma 1.

Lemma 2:
  ∀ q:lists, p:paths
    is_event_queue(q) ∧ count(q, x) = count(q[p], x)
                      ∧ q[p+<CAR, CAR>] = x ∧ is_int(x)
                      ∧ is_valid(p, q) ∧ is_all_cdrs(p) ∧ q[p] ≠ NIL
                      ∧ is_not_in(x, q, p)
  ⇒
    ∀y≠x(count(q', y) = count(q, y))
  where q' ≡ update(q, p, q[p+<CDR>]).

  The proof proceeds by induction on the length of p.

  Base case: p = <>
    Show:
      ∀ q:lists
        is_event_queue(q) ∧ count(q, x) = count(q[<>], x)
                          ∧ q[<>+<CAR, CAR>] = x ∧ is_int(x)
                          ∧ is_valid(<>, q) ∧ is_all_cdrs(<>)
                          ∧ q[<>] ≠ NIL ∧ is_not_in(x, q, <>)
      ⇒
      ∀y≠x(count(update(q, <>, q[p+<CDR>]), y) = count(q, y))
    The right-hand-side reduces to:
      ∀y≠x(count(cdr(q), y) = count(q, y))
    which reduces to:
      ∀y≠x(count(cdr(q), y) = if q = NIL then 0
                              else if caar(q) = y then cdar(q)
                              else count(cdr(q), y)
    which reduces to:
      ∀y≠x(count(cdr(q), y) = if false then 0
                              else if false then cdar(q)
                              else count(cdr(q), y)
```

```
  which reduces to:
    ∀y≠x(count(cdr(q), y) = count(cdr(q),y)
  which reduces to:
    true

Induction step:
  assume:
     ∀ q:lists
     is_event_queue(q) ∧ count(q, x) = count(q[p], x)
                        ∧ q[p↓<CAR, CAR>] = x ∧ is_int(x)
                        ∧ is_valid(p, q) ∧ is_all_cdrs(p)
                        ∧ q[p] ≠ NIL ∧ is_not_in(x, q, p)
   ⇒
     ∀y≠x(count(update(q, p, q[p↓<CDR>]), y) = count(q, y))
  Show:
   ∀ q:lists
     is_event_queue(q) ∧ count(q, x) = count(q[<f>↓p], x)
                        ∧ q[<f>↓p↓<CAR, CAR>] = x ∧ is_int(x)
                        ∧ is_valid(<f>↓p, q) ∧ is_all_cdrs(<f>↓p)
                        ∧ q[<f>↓p] ≠ NIL ∧ is_not_in(x, q, <f>↓p)
    ⇒
     ∀y≠x(count(update(q, <f>↓p, q[<f>↓p↓<CDR>]), y)
        = count(q, y))
  For the hypothesis is_all_cdrs(<f>↓p) to be true, f must be
  CDR.  Thus we must show:
   ∀ q:lists
     is_event_queue(q) ∧ count(q, x) = count(q[<CDR>↓p], x)
                        ∧ q[<CDR>↓p↓<CAR, CAR>] = x ∧ is_int(x)
                        ∧ is_valid(<CDR>↓p, q)
                        ∧ is_all_cdrs(<CDR>↓p) ∧ q[<CDR>↓p] ≠ NIL
                        ∧ is_not_in(x, q, <CDR>↓p)
    ⇒
     ∀y≠x(count(update(q, <CDR>↓p, q[<CDR>↓p↓<CDR>]), y)
        = count(q, y))
  The right-hand-side reduces to:
     ∀y≠x(count(cons(car(q),
                       update(cdr(q), p, cdr(q[p↓<CDR>]))), y)
        = count(q, y))
```



```
which reduces to:
if cons(car(q), update(cdr(q), p, cdr(q[p↓<CDR>])))= NIL
then 0
else if caar(cons(car(q),
                  update(cdr(q), p, cdr(q[p↓<CDR>])))) = y
   then cdar(cons(car(q),
                  update(cdr(q), p, cdr(q[p↓<CDR>]))))
   else count(cdr(cons(car(q),
                     update(cdr(q), p, cdr(q[p↓<CDR>])))), y)
 = if q = NIL then 0
   else if caar(q) = y then cdar(q)
   else count(cdr(q), y)
which reduces to:
   if false then 0
   else if car(car(q)) = y
      then cdr(car(q))
      else count(update(cdr(q), p, cdr(q[p↓<CDR>]))), y)
     = if false then 0
      else if caar(q) = y then cdar(q)
      else count(cdr(q), y)
which reduces to:
   if caar(q) = y then cdar(q)
   else count(update(cdr(q), p, (cdr(q))[p↓<CDR>]), y)
     = if caar(q) = y then cdar(q)
      else count(cdr(q), y)
which (if the antecedent of the induction hypothesis is true
when q is replaced by cdr(q)) reduces to:
   true
```

Thus it is necessary to show the following is true:

```
    is_event_queue(cdr(q))
      (By is_event_queue(q))
    ∧ count(cdr(q), x) = count((cdr(q))[p], x)
      (by count(q, x) = count(q[<CDR>↓p], x) ∧ is_not_in(x, q, p))
    ∧ (cdr(q))[p↓<CAR, CAR>] = x
      (by q[<CDR>↓p↓<CAR, CAR>] = x)
    ∧ is_int(x)
      (by is_int(x))
    ∧ is_valid(p, cdr(q))
      (by is_valid(<CDR>↓p, cdr(q)))
    ∧ is_all_cdrs(p)
      (by is_all_cdrs(<CDR>↓p))
    ∧ (cdr(q))[p] ≠ NIL
      (by q[<CDR>↓p] ≠ NIL)
    ∧ is_not_in(x, cdr(q), p)
      (by is_not_in(x, q, p))
```

End of Proof of Lemma 2.

End of Proof of (3).

<u>Proof of (2)</u>:

```
Let I ≡ is_event_queue(q) ∧ q0 = q ∧ count(q, x) = count(q[p], x)
                         ∧ is_valid(p, q) ∧ is_all_cdrs(p)
                         ∧ is_int(x) ∧ is_not_in(x, q, p)
and B ≡ q[p] ≠ NIL ∧ q[p↓<CAR, CAR>] ≠ x
```

then it is necessary to show:

$$I \wedge B \Rightarrow wp(\text{"p:= p}\downarrow\text{<CDR>"}, I)$$

which is equivalent to:

$$I \wedge B \Rightarrow I^{p}_{p\downarrow <CDR>}$$

show:

```
  is_event_queue(q) ∧ q0 = q ∧ count(q, x) = count(q[p], x)
                    ∧ is_valid(p, q) ∧ is_all_cdrs(p)
                    ∧ is_int(x) ∧ is_not_in(x, q, p)
                    ∧ q[p] ≠ NIL ∧ q[p↓<CAR, CAR>] ≠ x
⇒
  is_event_queue(q)
  ∧ q0 = q
  ∧ count(q, x) = count(q[p↓<CDR>], x)
  ∧ is_valid(p↓<CDR>, q)
  ∧ is_all_cdrs(p↓<CDR>)
  ∧ is_int(x)
  ∧ is_not_in(x, q, p↓<CDR>)
```

The right-hand-side reduces to:

```
  count(q, x) = count(q[p↓<CDR>], x)
    (which is true by count(q, x) = count(q[p], x) and
     the definition of count)
  ∧ is_valid(p↓<CDR>, q)
    (by is_valid(p, q) and q[p] ≠ NIL)
  ∧ is_all_cdrs(p↓<CDR>)
    (by is_all_cdrs(p) and the definition of ↓)
  ∧ is_not_in(x, q, p↓<CDR>)
    (by is_not_in(x, q, p))
```

End of Proof of (2).

<u>Proof of (1)</u>:

show:

```
is_event_queue(q) ∧ q0 = q ∧ is_int(x)
  ⇒
wp("p:= <>", is_event_queue(q) ∧ q0 = q ∧ count(q, x) = count(q[p], x)
                               ∧ is_valid(p, q) ∧ is_all_cdrs(p)
                               ∧ is_int(x) ∧ is_not_in(x, q, p))
```

The right-hand-side evaluates to:

```
    is_event_queue(q) ∧ q0 = q
                      ∧ count(q, x) = count(q[<>], x)
                      ∧ is_valid(<>, q)
                      ∧ is_all_cdrs(<>)
                      ∧ is_int(x)
                      ∧ is_not_in(x, q, <>)
```

which reduces to:

$$\begin{aligned} \text{is\_event\_queue}(q) \wedge\ & q_0 = q \\ \wedge\ & \text{count}(q, x) = \text{count}(q, x) \\ \wedge\ & \textbf{true} \\ \wedge\ & \textbf{true} \\ \wedge\ & \text{is\_int}(x) \\ \wedge\ & \textbf{true} \end{aligned}$$

which reduces to:

$$\begin{aligned} \text{is\_event\_queue}(q) \wedge\ & q_0 = q \\ \wedge\ & \text{count}(q, x) = \text{count}(q, x) \\ \wedge\ & \text{is\_int}(x)) \end{aligned}$$

which is implied by the hypothesis.

End of Proof of (1).

Note that it would be fairly simple to add to **post** a condition that $\text{caar}(q) = x$. Only the proof of S_5 would be changed.

APPENDIX B

A PATHS IMPLEMENTATION IN C

In this appendix we present some procedures and functions written in the programming language C that extend C by adding a facility for manipulating paths and <u>struct</u> variables. Since the programs are presented to give an idea about how paths are implemented, the uninteresting lower level functions have been omitted. The programs presented below assume that the following declarations have been made globally:

```
struct nodestr
{
    struct nodestr *carnode;
    struct nodestr *cdrnode;
    char speared;
    char valid;
    int ref_cnt;
    struct nodestr *back;
    int val;
    struct nodestr *next;
    char atom;
    char root;
    char suspend;
};

extern struct nodestr nodespace[FREESIZE];
extern struct nodestr *freehead, *Nilnode, *Invalid;

struct pathstr
{
    struct nodestr *termnode;
    char field;
    struct nodestr **owner;
};

typedef struct nodestr *value;
typedef struct pathstr *path;
typedef char field_sel;
typedef struct nodestr node;
typedef struct nodestr **structvar;
typedef struct nodestr *rnode;
```

Throughout the programs, compile-time constants are represented by identifiers entirely in upper-case (such as FREESIZE.)

```
/* return the value at the end of path p */
node *
extract ( p )
path p;
{
    validate ( p );
    return( field_extract(p → termnode, p → lastfield) );
}

/* change the value at the end of path p to newvalue */
update ( p , newvalue )
path p;
node *newvalue;
{
    node *oldvalue, *m;

    oldvalue = extrsct(p); /* extract will validate p */
    if ( is_speared(oldvalue) )
    {
        /* mark the path ends in the replaced value as "invalid" */
        treeinvalid(oldvalue);

        /* unspear all nodes up to the first fork node */
        m = (oldvalue → back);
        while ( !is_fork(m) )
        {
            unspear(m);
            m = (m → back);
        }
    }
    update_field( p → termnode, p → lastfield, newvalue );
    share( p → termnode, p → lastfield );
    dec_refcnt(oldvalue);
}

/* extend path p by field_selector f */
extend ( p , f )
path p;
field_sel f;
{
    if ( is_atom( extract(p) ) )
        error("extend atom"); /* extract validates p */
    dec_refcnt(p → termnode);
    spear(p → termnode, p → lastfield);
    (p → termnode) = extract_field(p → termnode, p → lastfield);
    inc_refcnt(p → termnode);
    (p → lastfield) = f;
}
```



```
/* retract path p by one field selector */
retract ( p )
path p;
{
    validate(p);
    if ( is_root(p → termnode) ) error("retract root");

    dec_refcnt(p → termnode);
    if (!has_speared_son( p → termnode) && !is_terminus(p → termnode))
        unspear(p → termnode);

    (p → lastfield) = fatherfield(p → termnode);
    (p → termnode) = ((p → termnode) → back);
    inc_refcnt(p → termnode);
}

/* s := f(s) -- leave paths alone (they point to the same nodes) */
shrink ( s , f )
structvar s;
field_sel f;
{
    node *root, *newtop, *oldtop;

    root = *s;
    oldtop = top(root);
    if ( is_atom(oldtop) ) error("shrink atom");

    newtop = extract_field(oldtop, f);

    /* invalidate paths in structure being thrown away */
        update_field(oldtop, f, Invalid); /* defer dec of newtop */
        treeinvalid(oldtop);

    if ( is_terminus ( oldtop ) ) /* fix up paths that were <f> to <> */
    {
        marksuspend(oldtop, TOP);
        update_field(oldtop, f, root); /* inc below */

        /* account for indirection to newroot from suspended oldtop */
            (root → ref_cnt) += (oldtop → ref_cnt) - 1;
    }

    root → top = newtop; dec_refcnt(oldtop); /* cancel def dec newtop */
    if ( is_speared(newtop) ) setbackptr(newtop, root);
}
```

```
/* value part of s := update(newtop, f, s) (leave paths alone) */
grow ( s , f , newtop )
structvar s;
field_sel f;
node *newtop;
{
    node *root, *oldtop, *placeholder;

    root = *s;
    oldtop = top(root);
    placeholder = extract_field(newtop, f);

    (root → top) = newtop;  share(root, TOP);  spear(root, TOP);
    update_field(newtop, f, oldtop);  dec_refcnt(placeholder);

    if ( is_speared(oldtop) )  setbackptr(oldtop, newtop);

    if ( is_terminus(root) ) /* fix up paths that were <> to <f> */
    {   node *newroot;

        newroot = makeroot();
        *s = newroot;  inc_refcnt(newroot); /* defer dec_refcnt(root) */
        (newroot → top) = newtop;

        /* account for indirection to newtop from suspended root */
            (newtop → ref_cnt) += (root → ref_cnt) - 1;

        setbackptr(newtop, newroot);
        marksuspend(root, f);
        dec_refcnt(root);  /* was deferred */
    }
}
```

APPENDIX C

A REAL TIME QUEUE IN PURE LISP

In this appendix we present LISP[1] programs that implement the real-time queue discussed in section 4.1. A queue is a nine-element list:

list[recopy, lendiff, #copy, H, T, h, H', T', H_R] ,

where we have given symbolic names to the components. For example, car[cdr[q]] is lendiff of queue q. An empty queue has value:

list[false, 0, 0, NIL, NIL, NIL, NIL, NIL, NIL] .

The queue functions Insert, Delete, and Query are given below. The auxiliary function Onestep is used, as its name implies, to perform one step of the reconstruction process.

```
Insert[q, v]≡
  if ¬ recopy ∧ lendiff > 0 →
        list[False, lendiff-1, 0, H, cons[v, T], NIL, NIL, NIL, NIL]
   □ ¬ recopy ∧ lendiff = 0 →
        Onestep[Onestep[True, 0, 0, H, cons[v, T], H, NIL, NIL, NIL]]
   □   recopy               →
        Onestep[Onestep[True, lendiff-1, #copy, H, T, h, H',
                             cons[v, T], H_R]]
  fi

Query[q]≡
  if ¬ recopy →  car[H]
   □   recopy →  car[h]
  fi
```

[1]For the sake of clarity, we have modified the syntax of LISP.

```
Delete[q]≡
   if ¬recopy ∧ lendiff > 0 →
         list[False, lendiff-1, 0, cdr[H], T, NIL, NIL, NIL, NIL]
    □ ¬recopy ∧ lendiff = 0 →
         Onestep[Onestep[True, 0, 0, cdr[H], T, cdr[H], NIL, NIL,
                        NIL]]
    □  recopy             →
         Onestep[Onestep[True, lendiff-1, #copy-1, H, T, cdr[h], H',
                        T', H_R]]
   fi

Onestep[q]≡
   if ¬recopy → q
    □  recopy ∧ ¬null[H] ∧ ¬null[T] →
         list[True, lendiff+1, #copy+1, cdr[H], cdr[T], h,
              cons[car[T], H'], T', cons[car[H], H_R]]

    □  recopy ∧  null[H] ∧ ¬null[T] →
        list[True, lendiff+1, #copy, NIL, NIL, h, cons[car[T], H'],
              T', H_R]

    □  recopy ∧  null[H] ∧  null[T] ∧ #copy > 1 →
         list[True, lendiff+1, #copy-1, NIL, NIL, h, cons[car[H_R], H'],
              T', cdr[H_R]]

    □  recopy ∧  null[H] ∧  null[T] ∧ #copy = 1 →
         list[False, lendiff+1, 0, cons[car[H_R], H'], T', NIL, NIL,
              NIL, NIL]

    □  recopy ∧  null[H] ∧  null[T] ∧ #copy = 0 →
         list[False, lendiff, 0, H', T', NIL, NIL, NIL, NIL]
   fi
```

In the second alternative of Onestep it is known that cdr[T] is NIL since T was exactly one longer than H when recopying started. Note that the only arithmetic operations performed are: add 1, subtract 1, and test for 0 or 1. They can be performed by appropriate list functions if we encode integers in unary notation. (The integer n is a list of n atoms.)