# Some Techniques for Proving Correctness of Programs which Alter Data Structures

R. M. Burstall

Department of Machine Intelligence
University of Edinburgh

## 1. INTRODUCTION

Consider the following sequence of instructions in a list-processing language with roughly ALGOL 60 syntax and LISP semantics (*hd* (*head*) is LISP CAR and *tl* (*tail*) is LISP CDR).

$x := cons(1, nil)$;
$y := cons(2, x)$;
$hd(x) := 3$; (compare LISP RPLACA)
$print(x)$; $print(y)$;

The intention is as follows.

$x$ becomes the list (1)
$y$ becomes the list (2, 1)
The head (first element) of the list $x$ becomes 3.

Since $y$ was manufactured from $x$ it 'shares' a list cell with $x$, and hence is side-effected by the assignment to $hd(x)$.

When $x$ is printed it is (3) and $y$ when printed is (2, 3) rather than (2, 1) as it would have been had the last assignment left it undisturbed.

How are we to prove assertions about such programs? Figure 1 traces the course of events in the traditional picture language of boxes and arrows. Our task will be to obtain a more formal means of making inferences, which, unlike the picture language, will deal with general propositions about lists. We will extend Floyd's proof system for flow diagrams to handle commands which process lists. The principles which apply to lists would generalise in a straightforward way to multi-component data structures with sharing and circularities.

Although this technique permits proofs, they are rather imperspicuous and fiddling for lack of appropriate higher level concepts. Investigating the special case of linear lists in more depth we define 'the list from $x$ to $y$' and consider systems of such lists (or perhaps we should say list fragments) which do not

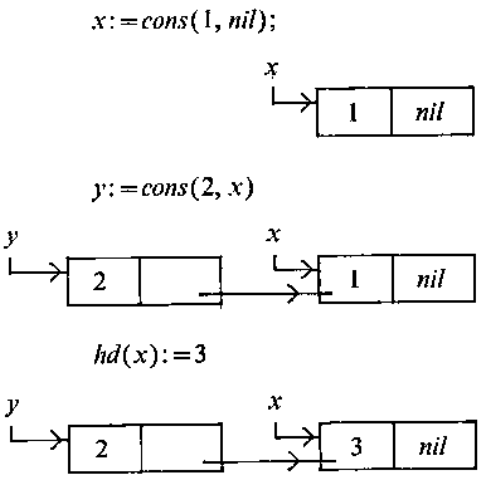



Figure 1.

share with each other or within themselves. (By a linear list we mean one which either terminates with *nil*, such as LISP $(A, (B, C), D)$, or is circular; by a tree we mean a list structure which terminates with atoms rather than with nil, such as LISP $((A \,.\, B) \,.\, (C \,.\, D))$). We thus get a rather natural way of describing the states of the machine and the transformations on them and hence obtain easy proofs for programs. Some ideas from the application of category theory to tree automata help us to extend this treatment from lists to trees: fragments of lists or trees turn out to be morphisms in an appropriate category. Acquaintance with category-theoretic notions is not however needed to follow the argument. Our aim has been to obtain proofs which correspond with the programmer's intuitive ideas about lists and trees. Extension to other kinds of data structures awaits further investigation.

## 2. PREVIOUS WORK

Since McCarthy (1963) raised the problem a number of techniques for proving properties of programs have been proposed. A convenient and natural method is due to Floyd (1967) and it has formed the basis of applications to non-trivial programs, for example by London (1970) and Hoare (1971). Floyd's technique as originally proposed dealt with assignments to numerical variables, for example, $x:=x+1$, but did not cater for assignments to arrays, for example, $a[i]:=a[j]+1$, or to lists, such as $tl(x):=cons(hd(x), tl(tl(x)))$. McCarthy and Painter (1967) deal with arrays by introducing 'change' and 'access' functions so as to write $a[i]:=a[j]+1$ as $a:=change(a, i, access$

$(a, j)+1$), treating arrays as objects rather than functions. King (1969) in mechanising Floyd's technique gives a method for such assignments which, however, introduces case analysis that sometimes becomes unwieldy. Good (1970) suggests another method which distinguishes by subscripts the various versions of the array. We will explain below how Good's method can be adapted to list processing. Although the proofs mentioned above by London (1970) and Hoare (1971) involve arrays they do not give rigorous justification of the inferences involving array assignments, which are rather straightforward.

List processing programs in the form of recursive functions have received attention from McCarthy (1963), Burstall (1969) and others, but quite different problems arise when assignments are made to components of lists. This was discussed in Burstall (1970) as an extension to the axiomatic semantics of ALGOL, but the emphasis there was on semantic definition rather than program proof.

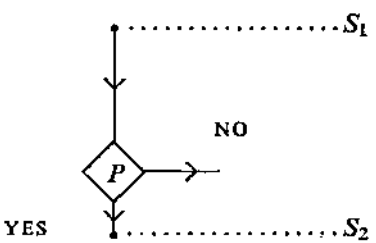
Hewitt (1970), Chapter 7, touches on proofs for list processing programs with assignments. J. Morris of Berkeley has also done some unpublished work, so have B.Wegbreit and J.Poupon of Harvard (Ph.D. thesis, forthcoming).

### 3. FLOYD'S TECHNIQUE

Let us recall briefly the technique of Floyd (1967) for proving correctness of programs in flow diagram form. We attach assertions to the points in the flow diagram and then verify that the assertion at each point follows from those at all the immediately preceding points in the light of the intervening commands. Floyd shows, by induction on the length of the execution path, that if this verification has been carried out whenever the program is entered with a state satisfying the assertion at the entry point it will exit, if at all, only with a state satisfying the assertion at the exit point.

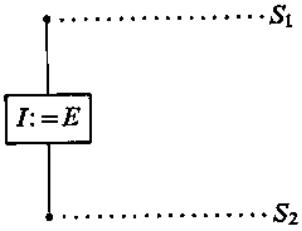
The rules for verification distinguish two cases: tests and assignments.

(1) A triple consisting of an assertion, a test and another assertion, thus




is said to be *verified* if $S_1, P \vdash_A S_2$, that is, assertion $S_2$ is deducible from $S_1$ and test $P$ using some axioms $A$, say the axioms for integer arithmetic. If $S_2$ is attached to the NO branch the triple is verified if $S_1, \neg P \vdash_A S_2$.

(2) A triple consisting of an assertion, an assignment and another assertion, thus

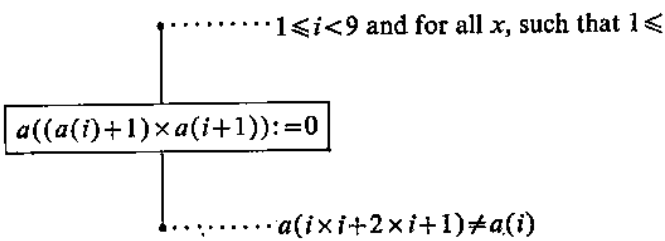



where $I$ is some identifier and $E$ some expression, is said to be *verified* if $S_1 \vdash_A [S_2]_I^E$ where $[S_2]_I^E$ means the statement $S_2$ with $E$ substituted for $I$ throughout. (This is called backward verification by King (1969); it is a variant of Floyd's original method, which introduces an existentially quantified variable.)

We will here retain the inductive method of Floyd for dealing with flow diagrams containing loops, but give methods for coping with more complex kinds of assignment command.

## 4. EXTENSION OF THE VERIFICATION PROCEDURE TO ARRAY ASSIGNMENTS

Consider the following command and assertions




The second assertion does hold if the first one does, but the verification rule given above for assignments to numeric variables, such as $j:=2\times j$, is inadequate for array assignments such as this. Thus attempts to substitute 0 for $a((a(i)+1)\times a(i+1))$ in $a(i\times i+2\times i+1)\neq a(i)$ merely leave it unchanged, but the unchanged assertion does not follow from the first assertion. (Floyd's version of the rule, using an existential quantifier is equally inapplicable.)

Following Good (1970), with a slightly different notation, we can overcome the difficulty by distinguishing the new version of the array from the old one by giving it a distinct symbol, say $a'$. We also make explicit the fact that other elements have not changed. We thus attempt to show that

$$1 \leqslant i \leqslant 9,\ (\forall x)(1 \leqslant x \leqslant 100 \Rightarrow a(x)=x),\ a'((a(i)+1)\times a(i+1))=0,$$
$$(\forall y)(y \neq (a(i)+1)\times a(i+1) \Rightarrow a'(y)=a(y)) \vdash_{\text{arith}} a'(i\times i+2\times i+1)\neq a'(i).$$

Once we note that $(a(i)+1)\times a(i+1)$ is $(i+1)^2$, as is $(i\times i+2\times i+1)$ and that, since $1 \leqslant i$, $(i+1)^2 \neq i$, we have $a'(i\times i+2\times i+1)=0$ and $a'(i)=i$, so

$a'(i)>0$. The distinction between $a$ and $a'$ and the condition that all elements which are not assigned to are unchanged reduce the problem to elementary algebra.

## 5. COMMANDS, CHANGE SETS AND TRANSFORMATION SENTENCES

The following technique is a variant of the one given by Good (1970). He uses it for assignments to arrays but not for list processing.

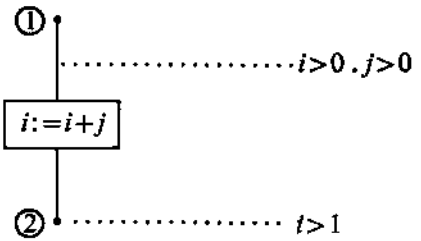
With each assignment command we associate a set of identifiers called the *change set* and a set of sentences called the *transformation sentences*. (Rules for specifying these are given below.) The basic rule of inference is:

If a set $S$ of sentences are written before a command $C$, and $C$ has a set $T$ of transformation sentences, then we may write a sentence $U$ after $C$ iff $S$, $T \vdash U'$, where $U'$ is $U$ with each identifier $i$ in the change set of $C$ replaced by $i'$.

(1) *Simple assignment*

| Command: $I:=E$ | e.g. $i:=i+j$ |
|---|---|
| Change set: $\{I\}$ | $\{i\}$ |
| Transformation sentences: $I'=E$ | $i'=i+j$ |

Example:




$i>1$ is legitimate at ② because $i>0, j>0, i'=i+j \vdash i'>1$.

*Note.* By $A \vdash B$ we mean $B$ is provable from $A$ using axioms of arithmetic or other axioms about the operations of the language.

*Note.* Variables (identifiers) in the program become *constants* in the logic.

(2) *Array assignment*

| Command: $A[E_1]:=E_2$ | e.g. $a[a[i]]:=a[i]+a[j]$ |
|---|---|
| Change set: $\{A\}$ | $\{a\}$ |
| Transformation sentences: | |
| $A'(E_1)=E_2$ | $a'(a(i))=a(i)+a(j)$ |
| $(\forall x)(x\neq E_1 \Rightarrow A'(x)=A(x))$ | $(\forall x)(x\neq a(i)\Rightarrow a'(x)=a(x))$ |

(3) *List processing*

| (a) Command: $hd(E_1):=E_2$ | e.g. $hd(tl(hd(i))):=hd(i)$ |
|---|---|
| Change set: $\{hd\}$ | $\{hd\}$ |
| Transformation sentences: | |
| $hd'(E_1)=E_2$ | $hd'(tl(hd(i)))=hd(i)$ |
| $(\forall x)(x\neq E_1\Rightarrow hd'(x)=hd(x))$ | $(\forall x)(x\neq tl(hd(i))\Rightarrow hd'(x)=hd(x))$ |

(b) Command: $tl(E_1):=E_2$ . . . as for *hd*

| | |
|---|---|
| (c) Command: $I:=cons(E_1, E_2)$ | e.g. $i:=cons(2, j)$ |
| Change set: $\{I, used, new\}$ | $\{i, used, new\}$ |
| Transformation sentences: | |
| $new' \notin used$ | $new' \notin used$ |
| $used' = used \cup \{new'\}$ | $used' = used \cup \{new'\}$ |
| $hd(new') = E_1$ | $hd(new') = 2$ |
| $tl(new') = E_2$ | $tl(new') = j$ |
| $I' = new'$ | $i' = new'$ |

*Note.* We assume $E_1$ and $E_2$ do not contain *cons*. Complex *cons* expressions must be decomposed.

It should be clear that the choice of two-element list cells is quite arbitrary, and exactly analogous rules could be given for a language allowing 'records' or 'plexes', that is a variety of multi-component cells.

## 6. CONCEPTS FOR LIST PROCESSING

We could now proceed to give examples of proofs for simple list processing programs, but with our present limited concepts it would be difficult to express the theorems and assertions except in a very *ad hoc* way. We want to talk about the list pointed to by an identifier *i* and say that this is distinct from some other list. We want to be able to define conveniently such concepts as reversing or concatenating (appending) lists.

To do this we now specialise our discussion to the case where *cons*, *hd* and *tl* are used to represent linear lists. Such lists terminate in *nil* or else in a cycle, and we do not allow atoms other than *nil* in the tail position. Thus we exclude binary trees and more complicated multi-component record structures.

First we define the possible states of a list processing machine.

A *machine* is characterised by:

$C$, a denumerable set of cells

*nil*, a special element

$A$, a set of atoms ($C$, $\{nil\}$ and $A$ are disjoint)

$\sigma$, a function from finite subsets of $C$ to $C$, such that $\sigma(X) \notin X$, a function for producing a new cell.

It is convenient to write $X^0$ for $X \cup \{nil\}$ where $X \subseteq C$.

A *state* is characterised by:

$U \subseteq C$, the cells in use

$hd$: $U \to A \cup U^0$

$tl$: $U \to U^0$ (thus we are talking about lists rather than binary trees)

Let $X^*$ be the set of all strings over a set $X$, including the empty string which we write as 1. Also let $T = \{true, false\}$.

It is convenient to define unit strings over $A \cup U^0$ thus

$Unit$: $(A \cup U^0)^* \to T$

$Unit(\sigma) \Leftrightarrow \sigma \in (A \cup U^0)$

We can now associate a set $\mathscr{L}$ of triples with a state. $(\alpha, u, v) \in \mathscr{L}$ means

that $\alpha$ is a list from $u$ to $v$ (or perhaps we should say a list fragment from $u$ to $v$). Thus

$$\mathscr{L} \subseteq (A \cup U^0)^* \times U^0 \times U^0$$

We shall write $u \xrightarrow{\alpha} v$ as a mnemonic abbreviation for $(\alpha, u, v)$, $hd\,u$ for $hd(u)$ and $tl\,u$ for $tl(u)$. We define $\mathscr{L}$ inductively, thus:

(i) $u \xrightarrow{1} u \in \mathscr{L}$ for each $u \in U^0$

(ii) if $u \xrightarrow{\alpha} v \in \mathscr{L}$ and $v \xrightarrow{\beta} w \in \mathscr{L}$ then $u \xrightarrow{\alpha\beta} w \in \mathscr{L}$

(iii) $u \xrightarrow{hd\,u} tl\,u \in \mathscr{L}$ for each $u \in U$.

For example, in figure 2, $x \xrightarrow{a_1a_2a_3} y$, $y \xrightarrow{a_4a_5a_3} y$, $y \xrightarrow{1} y$ and $z \xrightarrow{a_6a_7} nil$ are all in $\mathscr{L}$.

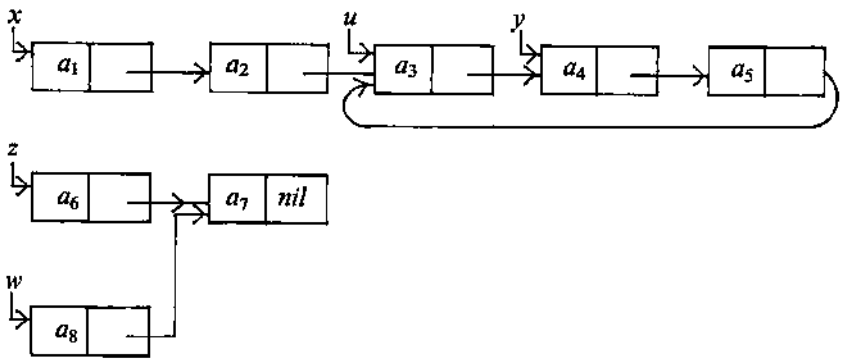



Figure 2.

Identity lists are defined thus

$$Identity: \mathscr{L} \to T$$

$$Identity(u \xrightarrow{\alpha} v) \Leftrightarrow \alpha = 1$$

A partial operation of composition $(\cdot)$ between lists is defined thus

$$\cdot : \mathscr{L} \times \mathscr{L} \to T$$

$$(u \xrightarrow{\alpha} v) \cdot (v \xrightarrow{\beta} w) = u \xrightarrow{\alpha\beta} w$$

Thus a list from $u$ to $v$ can be composed with one from $v'$ to $w$ if and only if $v = v'$.

The reader familiar with category theory will notice that $\mathscr{L}$ forms a category with $U$ as objects and the lists (triples in $\mathscr{L}$) as morphisms. (A definition of 'category' is given in the Appendix.) Indeed it is the free category generated by the graph whose arrows are the lists $u \xrightarrow{hd\,u} tl\,u$ for each $u$. There is a forgetful functor from $\mathscr{L}$ to $(A \cup U^0)^*$, where the latter is regarded as a category with one object. This functor gives the string represented by a list fragment (cf. Wegbreit and Poupon's notion of covering function in the unpublished work mentioned on p. 25).

We define the number of occurrences of a cell in a list by induction

$$\delta: U \times \mathscr{L} \to N$$

(i) $\delta_u(v \xrightarrow{1} v) = 0$

(ii) $\delta_u(x \xrightarrow{a} tl\, x \xrightarrow{\alpha} w) = \delta_u(tl\, x \xrightarrow{\alpha} w) + 1$ if $x = u$

$$= \delta_u(tl\, x \xrightarrow{\alpha} w) \quad \text{if } x \neq u$$

It follows by an obvious induction that $\delta_u(\lambda \cdot \mu) = \delta_u(\lambda) + \delta_u(\mu)$.

To keep track of the effects of assignments we now define a relation of distinctness between lists, that is, they have no cells in common.

$$Distinct: \mathscr{L} \times \mathscr{L} \to T$$

$Distinct(\lambda, \mu) \Leftrightarrow \delta_u(\lambda) = 0$ or $\delta_u(\mu) = 0$ for all $u \in U$.

We also define a property of non-repetition for lists

$$Nonrep: \mathscr{L} \to T$$

$Nonrep(\lambda) \Leftrightarrow \delta_u(\lambda) \leqslant 1$ for all $u \in U$

*Lemma* 1

(i) $Distinct(\lambda, u \xrightarrow{1} u)$ for all $\lambda$ and $u$

(ii) $Nonrep(u \xrightarrow{1} u)$ for all $u$

(iii) $Distinct(\lambda, \mu)$ and $Nonrep(\lambda)$ and $Nonrep(\mu) \Leftrightarrow Nonrep(\lambda \cdot \mu)$, if $\lambda \cdot \mu$ is defined.

*Proof.* Immediate.

We are now equipped to state the correctness criterion for a simple list processing program and to supply and prove the assertions. Consider the problem of reversing a list by altering the pointers without using any new space (figure 3). We first need to define an auxiliary function to reverse a string

$$rev: X^* \to X^*$$

$$rev(1) = 1$$

$$rev(x\alpha) = rev(\alpha)x \qquad \text{for } x \in X,\ \alpha \in X^*$$

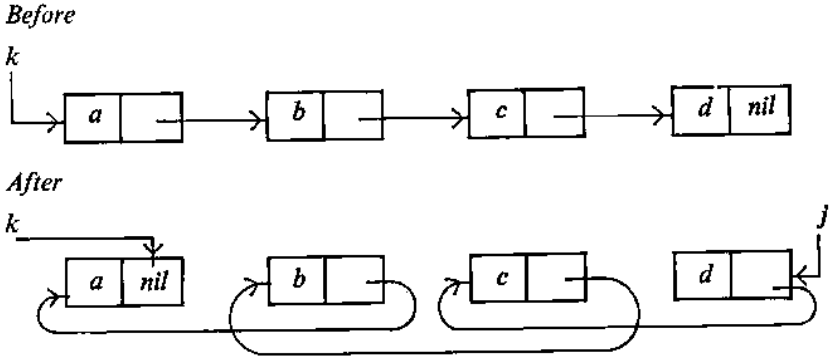



Figure 3.

$j = REVERSE(k)$

Assume $rev: U^* \to U^*$ reverses strings.

$K$ is a constant string.

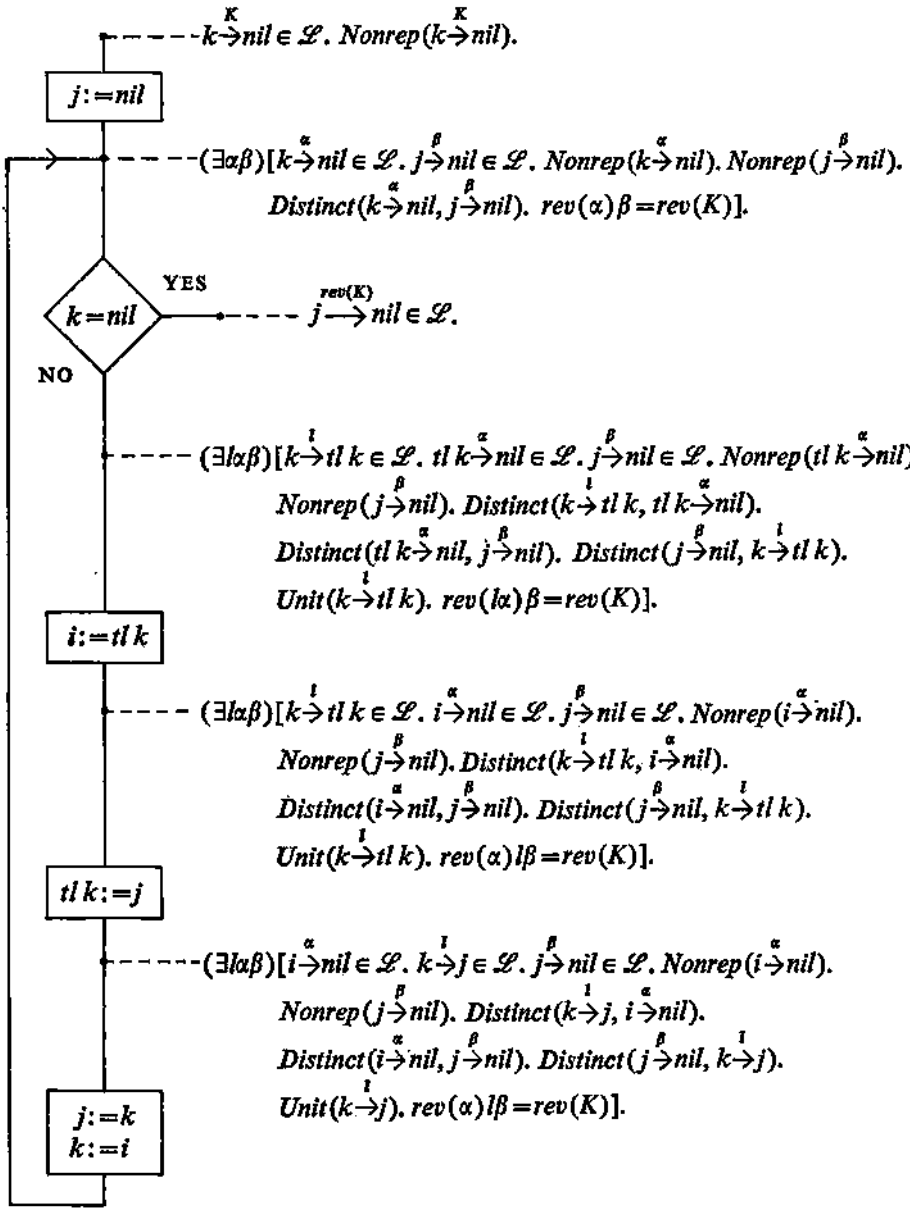



Figure 4. Reversing a list.

By well-known methods of induction on the length of the string (structural induction) we can prove simple lemmas such as

$$rev(\alpha\beta) = rev(\beta)\, rev(\alpha) \qquad \text{for } \alpha, \beta \in X^*$$

Notice the distinction between the function *rev* which works on strings and has easily proved properties and the function or procedure *REVERSE* to reverse lists using assignment, which we are about to define. The latter is essentially a function from machine states to machine states, where a machine state is characterised by the two functions *hd* and *tl*.

The flow diagram for *REVERSE* with assertions is given in figure 4. Notice that the assertions are long and tedious. We can verify the assertions by using the techniques given above, distinguishing between *tl* and *tl'* and consequently between *List*, *Distinct* and *List'*, *Distinct'*. The verification proofs are quite long for such a simple matter and very boring. We will not weary the reader with them; instead we will try to do better.

## 7. DISTINCT NON-REPEATING LIST SYSTEMS

We will now gather together the concepts introduced so far into a single notion, that of a Distinct Non-repeating List System (DNRL System). Using a suitable abbreviated notation we can render the assertions brief and perspicuous. To make the verification proofs equally attractive we show how the various kinds of commands, namely assignment to head or tail and *cons* commands, correspond to simple transformations of these systems. We can prove this once and for all using our previous technique and then use the results on a variety of programs.

We define a Distinct Non-repeating List System as an $n$-tuple of triples $\lambda_i$, $i = 1, \ldots, n$, such that

(i) $\lambda_i \in \mathscr{L}$ for each $i = 1, \ldots, n$

(ii) $Nonrep(\lambda_i)$ for each $i = 1, \ldots, n$

(iii) If $j \neq i$ then $Distinct(\lambda_i, \lambda_j)$ for each $i, j = 1, \ldots, n$

It is clear that if $S$ is a DNRL System then so is any permutation of $S$. Thus the ordering of the sequence is immaterial. We should not think of $S$ merely as a *set* of triples, however, since it is important whether $S$ contains a triple $x \overset{\alpha}{\rightarrow} y$ once only or more than once (in the latter case it fails to be a DNRL System unless $\alpha = 1$).

*Abbreviation.* We will write $u_1 \overset{\alpha_1}{\rightarrow} u_2 \overset{\alpha_2}{\rightarrow} u_3 \ldots u_{k-1} \overset{\alpha_{k-1}}{\rightarrow} u_k$, for $u_1 \overset{\alpha_1}{\rightarrow} u_2$, $u_2 \overset{\alpha_2}{\rightarrow} u_3$, $\ldots$, $u_{k-1} \overset{\alpha_{k-1}}{\rightarrow} u_k$.

We also write $*S$ for '$S$ is a DNRL System'.

For example, an abbreviated state description for the state shown in figure 2 is

$$*(x \overset{a_1 a_2}{\rightarrow} u \overset{a_3}{\rightarrow} y \overset{a_4 a_5}{\rightarrow} u,\ z \overset{a_6}{\rightarrow} tl\, z \overset{a_7}{\rightarrow} nil,\ w \overset{a_8}{\rightarrow} tl\, z)$$

or a less explicit description

$$(\exists \alpha\beta\gamma\delta)(*(x \overset{\alpha}{\rightarrow} u \overset{\alpha}{\rightarrow} y \overset{\beta}{\rightarrow} u,\ z \overset{\gamma}{\rightarrow} nil) \text{ and } *(w \overset{\delta}{\rightarrow} nil) \text{ and } Atom(a))$$

$j = REVERSE(k)$

Assume $rev: (A \cup U)^* \rightarrow (A \cup U)^*$ reverses strings.

$K$ is a constant string. Assume $a, b, c \ldots, \alpha, \beta, \gamma$ are existentially quantified before each assertion.

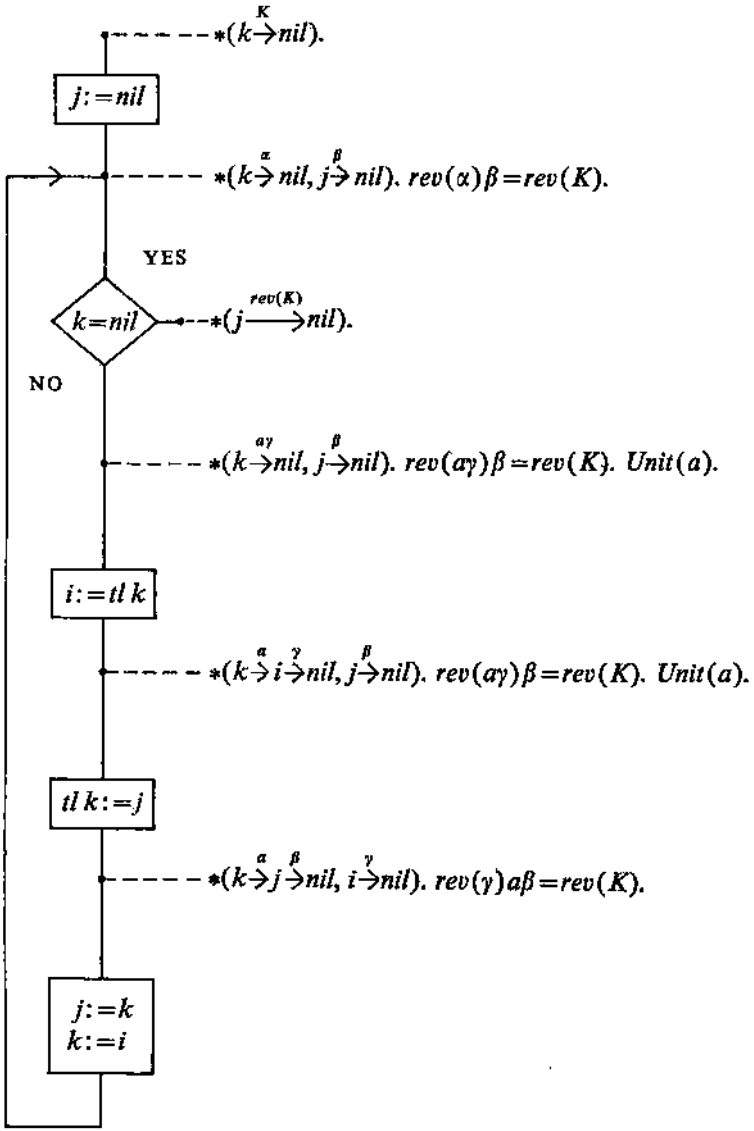



Figure 5. Reversing a list.

Figure 5 shows the *REVERSE* program again with much pithier assertions. We will now consider how to verify assertions such as these painlessly.

The following proposition which enables us to manipulate DNRL Systems follows easily from the definitions above.

*Proposition* 1

(i) *Permutation*
$*S \Rightarrow *S'$ if $S'$ is a permutation of $S$.

(ii) *Deletion*
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow *(\lambda_2, \ldots, \lambda_n)$

(iii) *Identity*
$$*(\lambda_1, \ldots, \lambda_n) \Rightarrow *(u \overset{1}{\rightarrow} u, \lambda_1, \ldots, \lambda_n)$$

(iv) *Composition*
$$*(u \overset{\alpha}{\rightarrow} v \overset{\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n) \Rightarrow *(u \overset{\alpha\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n)$$
and conversely
$$*(u \overset{\alpha\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n) \Rightarrow (\exists v)\ *(u \overset{\alpha}{\rightarrow} v \overset{\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n)$$

(v) *Distinctness*
$$*(u \overset{\alpha}{\rightarrow} v, u \overset{\beta}{\rightarrow} w) \Rightarrow \alpha = 1 \text{ or } \beta = 1$$

(vi) *Inequality*
$$*(u \overset{\alpha}{\rightarrow} v) \text{ and } u \neq v \Rightarrow (\exists b\beta)(\mathit{Unit}(b) \text{ and } \alpha = b\beta).$$

*Proof.* (i) and (ii) Immediate from the definition of *.

(iii) By Lemma 1 (i) and (ii).

(iv) By Lemma 1 (iii).

(v) If $\alpha \neq 1$ and $\beta \neq 1$ then $\delta_u(u \rightarrow v) = 1$ and $\delta_u(u \overset{\beta}{\rightarrow} w) = 1$ so they are not distinct.

(vi) By definition of $\mathscr{L}$.

We are now able to give the transformation sentences associated with the various commands, in terms of * rather than in terms of *hd* and *tl*. They enable us to verify the assertions in figure 5 very easily. The transformation sentences all involve replacing or inserting a component of a *-expression, leaving the other components unchanged. They are displayed in figure 6. We will make some comments on these transformation sentences and how to use them. Their correctness may be proved using the transformation sentences given earlier for *hd* and *tl* and the following lemma.

*Lemma* 2. Suppose for all $y \neq x$, $hd'y = hd\ y$ and $tl'y = tl\ y$. Then for all $\lambda \in \mathscr{L}$ such that $\delta_x(\lambda) = 0$ we have $\lambda \in \mathscr{L}'$ and $\delta'_y(\lambda) = \delta_y(\lambda)$ for all $y$.

*Corollary.* Under the same suppostion if $\lambda, \mu \in \mathscr{L}$ and $\delta_x(\lambda) = 0$ and $\delta_x(\mu) = 0$ then $\mathit{Nonrep}(\lambda) \Rightarrow \mathit{Nonrep}'(\lambda)$ and $\mathit{Distinct}(\lambda, \mu) \Rightarrow \mathit{Distinct}'(\lambda, \mu)$.

*Proof.* By an obvious induction on $\mathscr{L}$.

The rule for assignment to a simple identifier is as before.

Transformation sentences are given for the YES and NO branches of a test, even though these do not alter the state (their change sets are empty).

Assume $n, \lambda_1, \ldots, \lambda_n, x, y, a$ are *universally* quantified. The transformation sentences are shown to the right of the command. We assume that $E, E_1$ and $E_2$ do not contain *cons*.

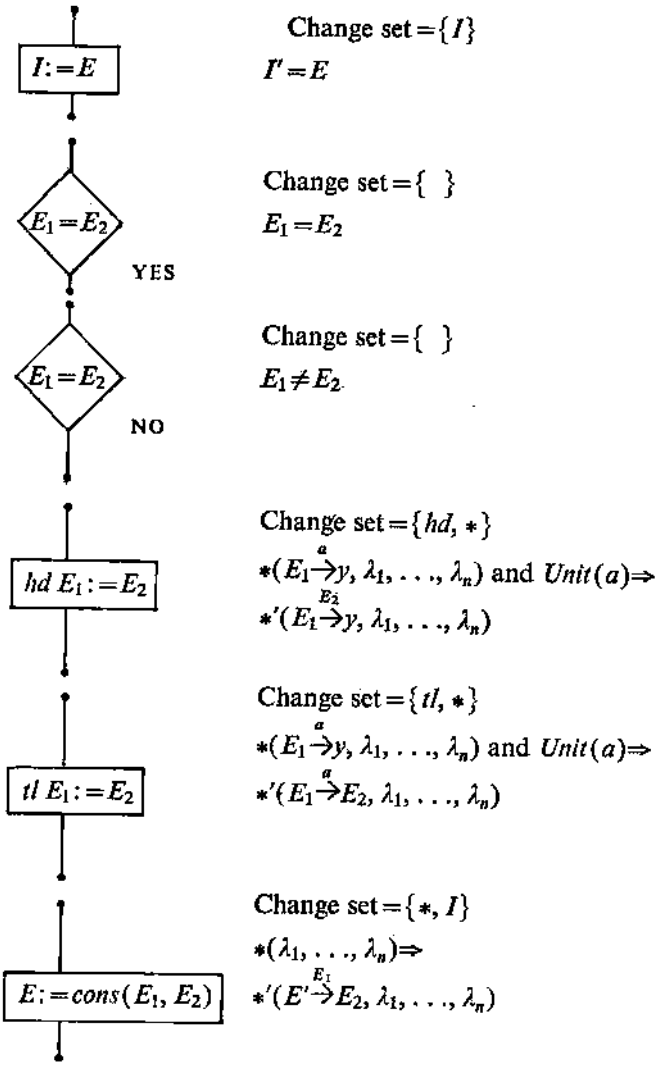



Figure 6. Transformation sentences for state descriptions.

Consider for example the NO branch of the test '$k=nil$' in the *REVERSE* program of figure 5.

Before the command we have

$$(\exists\alpha\beta)[*(k\overset{\alpha}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(\alpha)\beta=rev(K)] \tag{1}$$

The transformation sentence, putting $n=1$, is

$$k\neq nil. \tag{2}$$

After the command, we have

$$(\exists\alpha\gamma\beta)[*(k\overset{\alpha\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(\alpha\gamma)\beta=rev(K).\ Unit(a)] \tag{3}$$

But (3) follows immediately from (1) and (2) using Proposition 1 (vi).

In general, Proposition 1 will be used to reorder or otherwise manipulate the *-expressions and if $E_1$ or $E_2$ contain references to *hd* or *tl* these will need to be removed by replacing them by $\hat{E}_1$ and $\hat{E}_2$ using Proposition 2. Still, the verification is quite trivial.

Consider another example from the *REVERSE* program, the command '*tl* $k:=j$'.

Before the command, we have

$$(\exists\alpha\gamma\beta)[*(k\overset{\alpha}{\rightarrow}i\overset{\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(\alpha\gamma)\beta=rev(K).\ Unit(a)] \tag{1}$$

The transformation sentence, putting $n=2$, is

$$*(k\overset{a}{\rightarrow}y, \lambda_1, \lambda_2)\Rightarrow$$
$$*'(k\overset{a}{\rightarrow}j, \lambda_1, \lambda_2), \text{ for all } a, y, \lambda_1, \lambda_2 \tag{2}$$

Rewriting the statement after the command with *' for *, we have

$$(\exists\alpha\gamma\beta)[*'(k\overset{\alpha}{\rightarrow}j\overset{\beta}{\rightarrow}nil, i\overset{\gamma}{\rightarrow}nil).\ rev(\gamma)\alpha\beta=rev(K)] \tag{3}$$

We must prove this from (1) and (2).

Combining (1) with (2) we get

$$(\exists\alpha\gamma\beta)[*'(k\overset{\alpha}{\rightarrow}j, i\overset{\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(\alpha\gamma)\beta=rev(K).\ Unit(a)]$$

But permuting the *' expression (Proposition 1 (*i*)) and using obvious properties of *rev* we get (3).

The sentences for *hd* and *cons* are used in an analogous way.

Because their meaning changes in a relatively complex way it is advisable to debar *hd* and *tl* from appearing in state descriptions and work in terms of * alone. We now consider how to reduce an expression involving *hd* or *tl* with respect to a state description so as to eliminate references to these. For example the expression $hd(tl(tl(i)))$ with respect to the state description $*(i\overset{a}{\rightarrow}x\overset{b}{\rightarrow}j\overset{c}{\rightarrow}y, \ldots)$ with $Unit(a)$, $Unit(b)$, $Unit(c)$ reduces to $c$. If such an expression occurs in a transformation sentence we reduce it to obtain an equivalent transformation sentence not involving *hd* or *tl*.

The reduction of an expression $E$ with respect to a state description $D$, written $\hat{E}$, is defined recursively by

(i) If $E$ is an identifier, a constant or a variable then $\hat{E}=E$

(ii) If $E$ is *hd* $E_1$ and $D$ contains $*(\hat{E}_1\overset{a}{\rightarrow}x, \ldots)$ and $Unit(a)$ then $\hat{E}=a$

(iii) If $E$ is $tl\,E_1$ and $D$ contains $*(\hat{E}_1 \overset{a}{\rightarrow} x, \ldots)$ and $Unit(a)$ then $\hat{E}=x$.

*Proposition* 2. $D \vdash \hat{E}=E$.

The proof is straightforward by induction on the structure of $E$, using the definition of *.

## 8. EXAMPLES OF PROOFS FOR LIST PROCESSING

The transformation sentences can best be understood by seeing how we use them in proving some simple list processing programs which alter the list structures. Although these programs are short their mode of operation, and hence their correctness, is not immediately obvious. We have already looked at the program *REVERSE*.

Another example, involving *cons*, is concatenation of two lists, copying the first and terminating it with a pointer to the second, see figure 7. Notice that the input lists need not necessarily be distinct. The other, destructive, method of concatenation is to overwrite the final *nil* of the first list with a pointer to the second. In this case our initial condition would have to be $*(k \overset{K}{\rightarrow} nil, l \overset{L}{\rightarrow} nil)$, ensuring distinctness.

Our next example, figure 8, is reversing a cyclic list, where instead of terminating with *nil* the list eventually reaches the starting cell again.

The next example, figure 9, involves a list with sub-lists. A list of lists are to be concatenated together and the *REVERSE* and *CONCAT* routines already defined are used. The suspicious reader may notice that we are making free with '. . .' in the assertions, but we can always replace $(s_1, \ldots, s_n)$ by, say, $(s_i)_{i=1}^{n}$ or $sequence(s, 1, n)$, so nothing mysterious is involved.

Some of the attractions of such proofs seems to come from the fact that the form of the assertions is graph-like and so (at least in figures 6, 7 and 8) strictly analogous to the structures they describe.

## 9. TREES

We will now pass from lists to trees. We will consider general 'record' or 'plex' structures with various kinds of cell each having a specified number of components, as for example in Wirth and Hoare (1966). A particular case is that of binary trees. Thus, instead of

$$hd\colon U \rightarrow A \cup U^0$$
$$tl\colon U \rightarrow U^0$$

we now allow

$$hd\colon U \rightarrow A \cup U$$
$$tl\colon U \rightarrow A \cup U$$

(*nil* has no special place in binary trees; it can be considered an atom like any other).

More generally we may replace trees built using *cons* with trees (terms or expressions) built using any number of operators with arbitrary numbers of arguments, corresponding to different kinds of records.

$m = CONCAT(k, l)$
$K$ and $L$ are constant strings of atoms, $a$ and $b$ units.

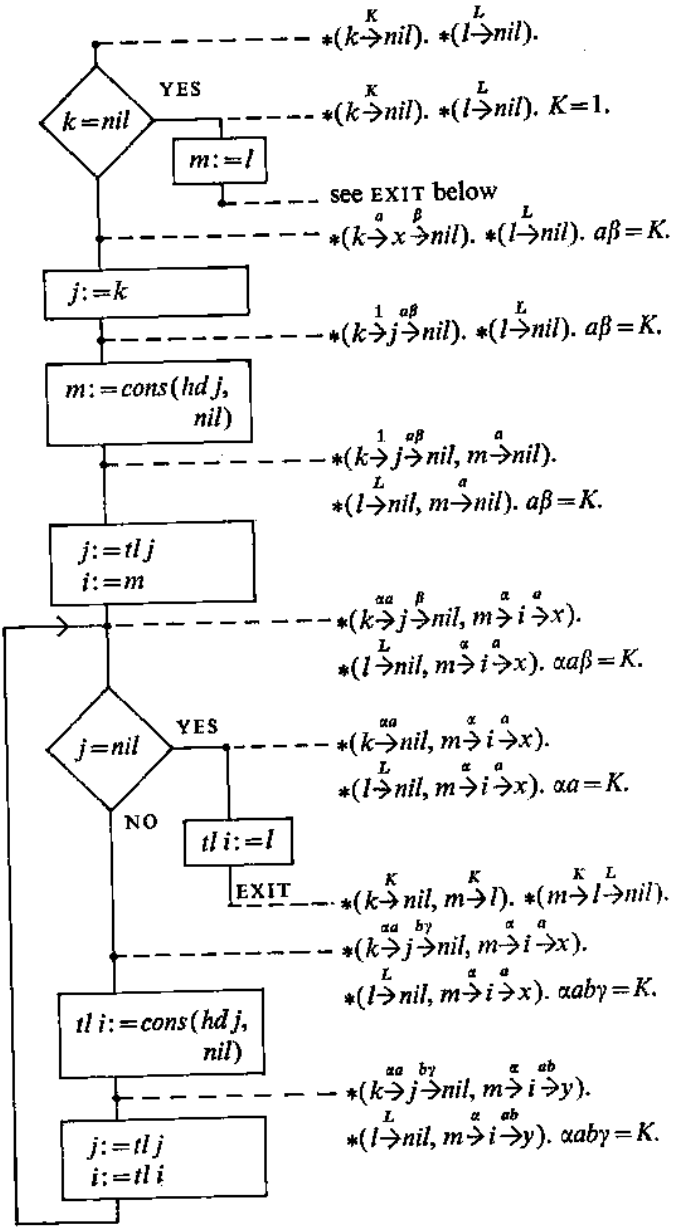



Figure 7. Concatenation.

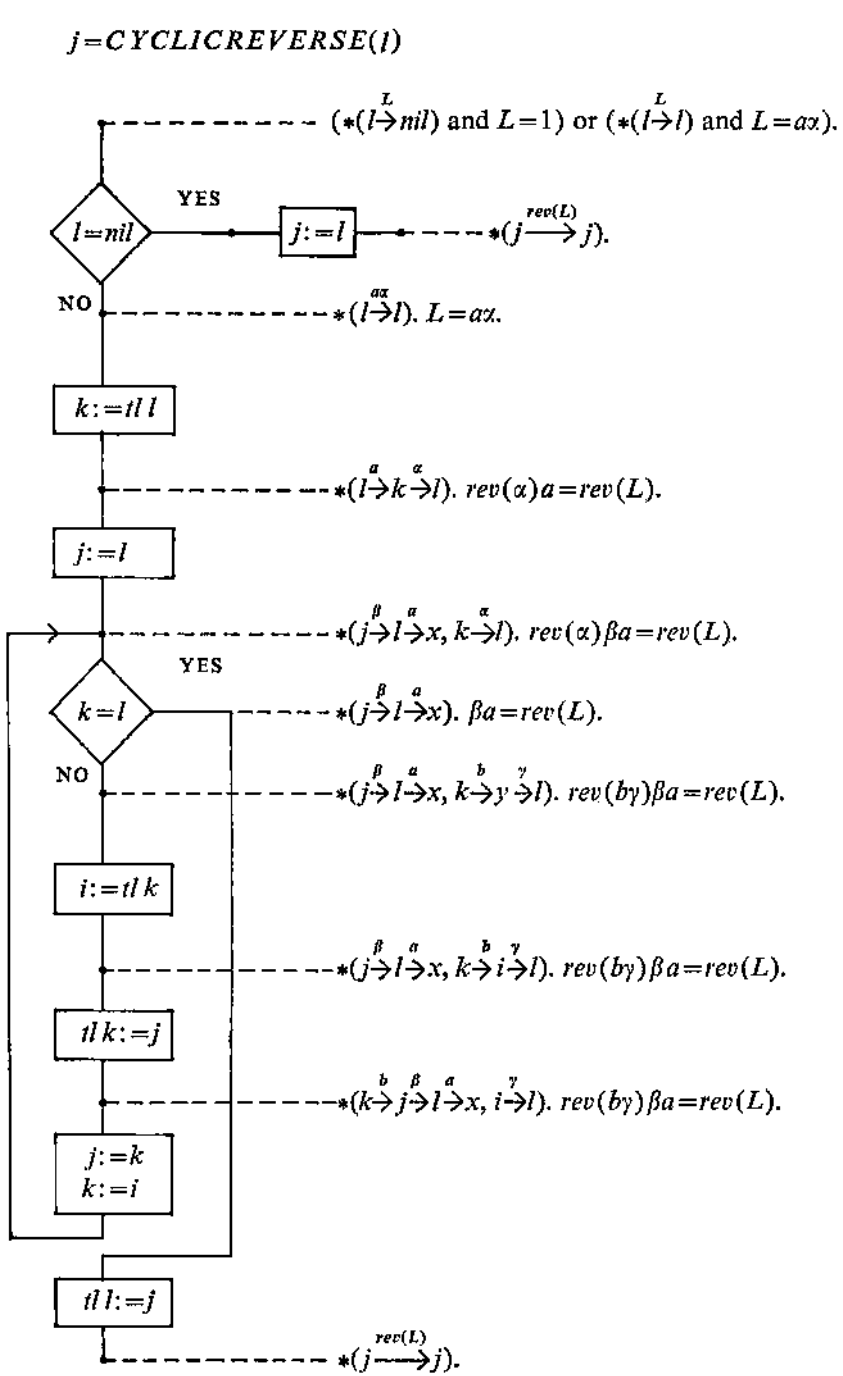



Figure 8. Reversing a cyclic list.

$l = MULTICONCAT(i)$

Assume $N, C_1, \ldots, C_N, A_1, \ldots, A_N$ are constants.

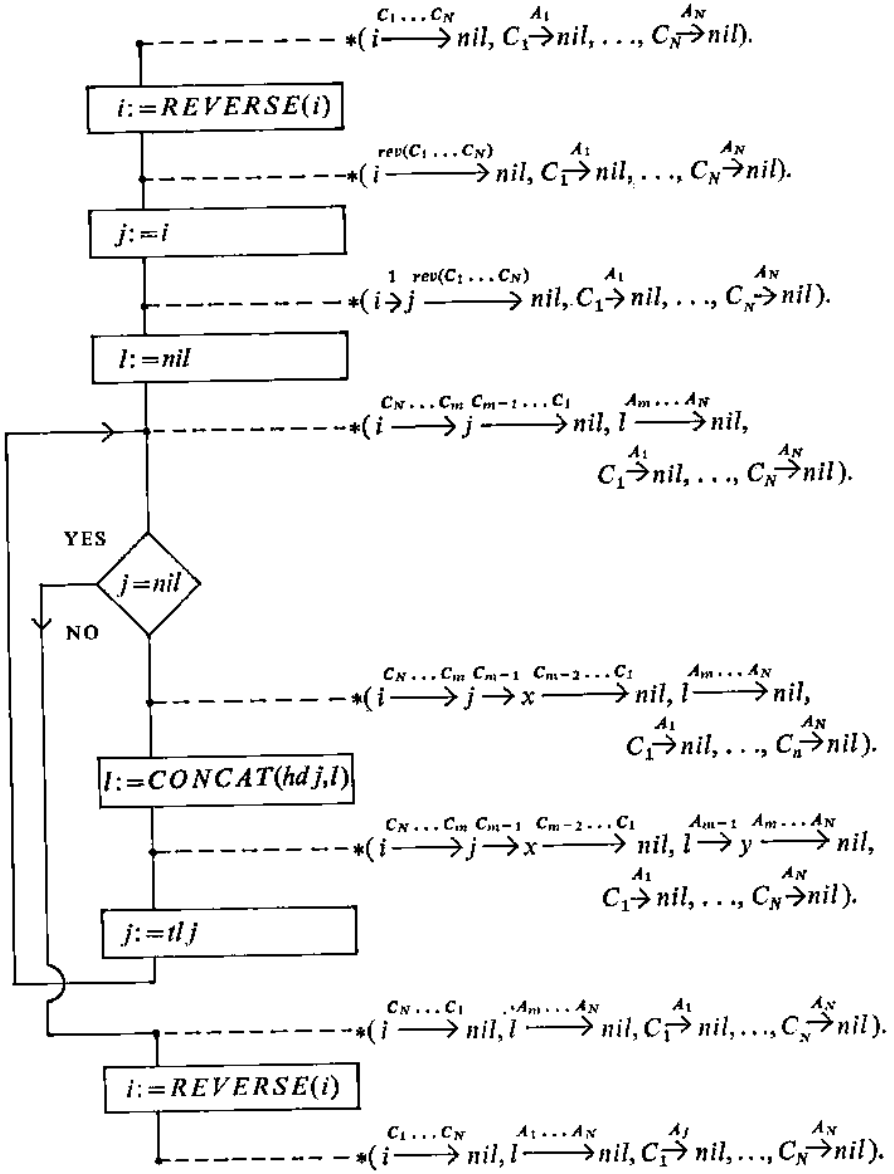



Figure 9. Multiple concatenation.

Our previous discussion depended rather heavily on the concatenation operation for lists. Fortunately Lawvere (1963) with his notion of a 'Free Theory' has shown how concatenation can be extended from strings to trees (see also Eilenberg and Wright 1967). His category theory approach is convenient in view of the remark in an earlier section that $\mathscr{L}$ forms a category. We will not present our ideas in category-theoretic language, since this may not be familiar to some readers, but will be content to point out where the concept of category is appropriate.

Our first task is to define a system of trees or expressions with a composition operation, analogous to the strings used above. We will then apply them to tree-like data structures under assignment commands.

Let $\Omega=\{\Omega_n\}$, $n=0, 1, \ldots$ be a sequence of sets of operators, $\{x_i\}$, $i=1, 2, \ldots$ a sequence of distinct variables, and let $X_m=\{x_1, \ldots, x_m\}$.

We define a sequence of sets of terms (or trees) $T=\{T_\Omega(X_m)\}$, $m=0, 1, \ldots$. $T_\Omega(X_m)$ means all terms, in the usual sense of logic, in the operators $\Omega$ and variables $X_m$. We can regard $T_\Omega(X_m)$ as a subset of the strings $(\bigcup\Omega\cup X_m)^*$ and define it inductively thus

(i) $x\in T_\Omega(X_m)$ if $x\in X_m$

(ii) $\omega t_1 \ldots t_n\in T_\Omega(X_m)$ if $\omega\in\Omega_n$ for some $n$ and $t_1, \ldots, t_n\in T_\Omega(X_m)$.

(Algebraically speaking $T_\Omega(X_m)$ forms the word algebra or generic algebra over $\Omega$ with generating set $X_m$.)

If $t\in T_\Omega(X_m)$ and $(s_1, \ldots, s_m)\in T_\Omega(X_n)^m$, $m\geqslant 0$, by the composition $t\cdot(s_1, \ldots, s_m)$ we mean the term in $T_\Omega(X_n)$ obtained by substituting $s_i$ for $x_i$ in $t$, $i=1, \ldots, m$.

To preserve the analogy with our previous treatment of strings we would like to make composition associative. For this we consider $n$-tuples of trees. Thus, more generally, if $(t_1, \ldots, t_l)\in T_\Omega(X_m)^l$ and $(s_1, \ldots, s_m)\in T_\Omega(X_n)^m$, we define the composition by

$$(t_1, \ldots, t_l)\cdot(s_1, \ldots, s_m)=(t_1\cdot(s_1, \ldots, s_m), \ldots, t_l\cdot(s_1, \ldots, s_m)).$$

This composition is in $T_\Omega(S_m)^l$.

For example if $\Omega_2=\{cons\}$, $\Omega_0=\{a, b, c, \ldots\}$, and we allow ourselves to write parentheses in the terms for readability

$$(cons(x_1, a), cons(x_2, x_1), b)\in T_\Omega(X_2)^3$$
$$(c, x_1)\in T_\Omega(X_1)^2$$

and their composition is

$$(cons(c, a), cons(x_1, c), b)\in T_\Omega(X_1)^3.$$

The composition is now associative and $(x_1, \ldots, x_n)\in T_\Omega(X_n)^n$ is an identity for each $n$. It is, however, a partial operation (compare matrix multiplication, which is only defined for matrices of matching sizes).

We see now that the disjoint union of the $T_\Omega(X_n)$ forms a category, $T_\Omega$, with as objects the integers $n=0, 1, \ldots$ and with a set of morphisms $T_\Omega(X_n)^m$ from $m$ to $n$ for each $m$, $n$. Indeed, this is just what Lawvere calls the Free Theory on $\Omega$. Eilenberg and Wright (1967) use it to extend automata theory to trees as well as strings, giving a category theory presentation (see

also Arbib and Giv'eon (1968)). The category $T_\Omega$ replaces the monoid $(A \cup U)^*$ used in our treatment of lists, the main difference being that the composition operation, unlike string concatenation, is only partial. The strings over an alphabet $\Sigma$ can be regarded as a special case by taking $\Omega_1$ to be $\Sigma$ and $\Omega_0$ to be $\{nil\}$. In the Appendix we summarise the abstract definition of a Free Theory as given by Eilenberg and Wright. We will not use any category theory here, but it is perhaps comforting to know what kind of structure one is dealing with.

We will consider a fixed $\Omega$ and $X$ for the time being and write $T_{mn}$ for $T_\Omega(X_n)^m$, the set of $m$-tuples of terms in $n$ variables. If $\omega \in \Omega_n$ we will take the liberty of writing $\omega$ for the term $\omega x_1 \ldots x_n$ in $T_{1n}$. This abbreviation may appear more natural if we think of an element $\tau$ of $T_{1n}$ as corresponding to the function $\lambda x_1 \ldots x_n \cdot t$ from $T_\Omega(\varnothing)^n$ to $T_\Omega(\varnothing)$. For example *cons* is short for $\lambda x_1, x_2 \cdot cons(x_1, x_2)$, but not of course for $\lambda x_1, x_2 \,.\, cons(x_2, x_1)$.

We will write 1 for the identity $(x_1, \ldots, x_n) \in T_{nn}$ and 0 for the 0-tuple $() \in T_{0n}$. We will sometimes identify the 1-tuple $(x)$ with $x$.

## 10. STATE DESCRIPTIONS USING TREES

We can now use the $m$-tuples of terms, $T_{mn}$, just as we previously used strings to provide state descriptions.

Suppose now that each $\omega \in \Omega_n$, $n=0, 1, \ldots$, corresponds to a class of cells $U_\omega$ (records) with $n$-components and that these components can be selected by functions

$$\delta_i^\omega \colon U_\omega \to \bigcup \{U_\omega : \omega \in \bigcup \Omega\}, \quad i=1, \ldots, n$$

We put $U = \bigcup \{U_\omega : \omega \in \bigcup \Omega\}$.

For example if $\Omega_2 = \{cons\}$, $\Omega_0 = A$ and $\Omega_i = \varnothing$ for $\omega \neq 0$ or 2 then

$$\delta_1^{cons} \text{ is } hd \colon U_{cons} \to U_{cons} \bigcup A$$
$$\delta_2^{cons} \text{ is } tl \colon U_{cons} \to U_{cons} \bigcup A$$

(here we have put $U_a = \{a\}$ for each $a \in A$, and $U_{cons}$ is just our previous $U$, the set of list cells).

A state is defined by the $U_\omega$ and the $\delta_i^\omega$ and we associate with it a set $\mathscr{T}$ of triples, just as the set $\mathscr{L}$ was previously associated with a state.

If a triple $(\tau, u, v)$ is in $\mathscr{T}$ then for some $m, n$, $u \in U^m$, $v \in U^n$ and $\tau \in T_{mn}$. As before we write $u \xrightarrow{\tau} v$ for $(\tau, u, v)$. We define $\mathscr{T}$ inductively thus:

(i) If $\tau \in T_{mn}$ is $(x_{i_1}, \ldots, x_{i_m})$ and $i_j \in \{1, \ldots, n\}$ for $j=1, \ldots, m$ (that is, $\tau$ involves only variables and not operators) then $(u_{i_1}, \ldots, u_{i_m}) \xrightarrow{\tau} (u_1, \ldots, u_n)$ is in $\mathscr{T}$.

(ii) If $u \xrightarrow{\tau} v \in \mathscr{T}$ and $v \xrightarrow{\sigma} w \in \mathscr{T}$ then their composition $(u \xrightarrow{\tau} v) \cdot (v \xrightarrow{\sigma} w)$ is defined as $u \xrightarrow{\tau \cdot \sigma} w$, and it is in $\mathscr{T}$.

(iii) If $(u_1) \xrightarrow{(t_1)} v, \ldots, (u_n) \xrightarrow{(t_n)} v$ are in $\mathscr{T}$ then $(u_1, \ldots, u_n) \xrightarrow{(t_1, \ldots, t_n)} v$ is in $\mathscr{T}$.

(iv) If $\omega \in \Omega_n$ and $\delta_i^\omega(u) = v_i$ for $i=1, \ldots, n$ then $(u) \xrightarrow{\omega} (v_1, \ldots, v_n)$ is in $\mathscr{T}$.

Taking the case where $\Omega$ consists of *cons* and atoms, (iv) means that $(u) \xrightarrow{cons} (v, w) \in \mathscr{T}$ if $hd(u)=v$ and $tl(u)=w$, also that $(a) \xrightarrow{a} ( )$ is in $\mathscr{T}$ for each atom $a \in A$.

An example will make this clearer. In the state pictured in figure 10 $\mathscr{T}$ contains the following triples, amongst others,

(i) $(i) \xrightarrow{cons(cons(cons(c,d),a),cons(cons(c,d),b))} ( )$

(ii) $(i) \xrightarrow{cons(cons(x_1,a),cons(x_1,b))} (j)$

(iii) $(j) \xrightarrow{cons(c,d)} ( )$

(iv) $(i) \xrightarrow{cons(cons(x_1,a),x_2)} (j, k)$

(v) $(l) \xrightarrow{cons(x_1,x_1)} (l)$

The first of these is the composition of the second and third.

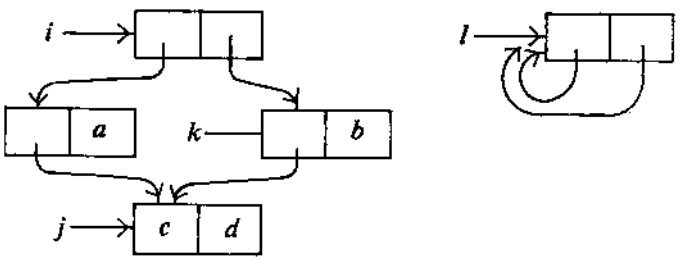



Figure 10.

It will be noticed that $\mathscr{T}$ forms a category with objects $u$ for each $u \in U^n$, $n=0, 1, \ldots$. The triples $(\tau, u, v)$ in $\mathscr{T}$ are the morphisms from $u$ to $v$. There is a forgetful functor *represents*: $\mathscr{T} \to T$.

We can now define a Distinct Non-repeating Tree System analogous to a Distinct Non-repeating List System. We take over the definitions of Distinctness and Non-repetition almost unchanged except that they now apply to $\mathscr{T}$ rather than to $\mathscr{L}$.

We call $(\tau, u, v) \in \mathscr{T}$ *elementary* if $\tau$ involves variables and operators in $\Omega_0$ but none in $\Omega_n$, $n \geqslant 1$ (for example, trees which share only elementary constituents can share atoms but not list cells).

We now define, as for lists, the number of occurrences of a cell in an $n$-tuple of trees, thus

$$\delta: U \times \mathscr{T} \to N$$

(i) $\delta_u(v \xrightarrow{\tau} w)=0$ if $u$ is elementary

(ii) $\delta_u((x) \xrightarrow{\omega} v \xrightarrow{\tau} w) = \delta_u(v \xrightarrow{\tau} w)+1$ if $x=u$
$= \delta_u(v \xrightarrow{\tau} w)$ if $x \neq u$ } for $\omega \in \Omega_n$, $n>0$

(iii) $\delta_u((u_1, \ldots, u_n) \xrightarrow{(\tau_1, \ldots, \tau_n)} v) = \delta_u((u_1) \xrightarrow{\tau_1} v) + \cdots + \delta_u((u_n) \xrightarrow{\tau_n} v)$

Notice that for $a \in \Omega_0$ we have $\delta_u((a) \xrightarrow{a} ()) = 0$, since we do not wish to notice sharing of atoms, which is innocuous.

The definitions of *Distinct* and *Nonrep* are as before replacing $\mathscr{L}$ by $\mathscr{T}$. Lemma 1 holds unchanged with the obvious addition that Distinctness and Non-repetition are preserved by formation of $n$-tuples in the same way as by composition.

The definition of a Distinct Non-repeating Tree System is, as before, a $k$-tuple of elements of $\mathscr{T}$, $\lambda_i$, $i = 1, \ldots, k$, such that

(i) $\lambda_i$ is in $\mathscr{T}$ for $i = 1, \ldots, k$

(ii) *Nonrep*$(\lambda_i)$ for each $i = 1, \ldots, n$

(iii) If $j \neq i$ then *Distinct*$(\lambda_i, \lambda_j)$ for each $i, j \in 1, \ldots, k$.

We employ the same abbreviation as before, writing $*S$ for '$S$ is a Distinct Non-repeating Tree System'.

We can adapt Proposition 1 and specify some useful properties of such systems.

*Proposition* 3.

(i) *Permutation*

$*S \Rightarrow *S'$ if $S'$ is a permutation of $S$.

(ii) *Deletion*

$*(\lambda_1, \ldots, \lambda_k) \Rightarrow *(\lambda_2, \ldots, \lambda_k)$.

(iii) *Rearrangement of variables*

$$*(\lambda_1, \ldots, \lambda_k) \Rightarrow *((u_{i_1}, \ldots, u_{i_m}) \xrightarrow{(x_{i_1}, \ldots, x_{i_m})} (u_1, \ldots, u_n), \lambda_1, \ldots, \lambda_k)$$

if $i_j \in \{1, \ldots, n\}$ for $j = 1, \ldots, m$.

(iv) *Composition*

$$*(u \xrightarrow{\tau} v \xrightarrow{\sigma} w, \lambda_1, \ldots, \lambda_k) \Leftrightarrow *(u \xrightarrow{\tau \cdot \sigma} w, \lambda_1, \ldots, \lambda_k)$$

and conversely

$$*(u \xrightarrow{\tau \cdot \sigma} w, \lambda_1, \ldots, \lambda_k) \Rightarrow (\exists v)\, *(u \xrightarrow{\tau} v \xrightarrow{\sigma} w, \lambda_1, \ldots, \lambda_k)$$

(v) *Tupling*

$$*((u_1) \xrightarrow{\tau_1} v, \ldots, (u_m) \xrightarrow{\tau_m} v, \lambda_1, \ldots, \lambda_k)$$
$$\Leftrightarrow *((u_1, \ldots, u_m) \xrightarrow{(\tau_1, \ldots, \tau_m)} v, \lambda_1, \ldots, \lambda_k) \quad (m \geqslant 0).$$

(vi) *Distinctness*

$$*(u \xrightarrow{\tau} v, u \xrightarrow{\sigma} w) \Rightarrow u \xrightarrow{\tau} v \text{ is elementary}$$
$$\text{or } u \xrightarrow{\sigma} w \text{ is elementary.}$$

(vii) *Inequality*

$*((u) \xrightarrow{\tau} (v_1, \ldots, v_n))$ and $u \neq v_i$, $i = 1, \ldots, n$
and $u \in U_\omega \Rightarrow (\exists \sigma)(\tau = \omega \cdot \sigma)$.

*Proof.* Obvious using Lemma 1 adapted for trees.

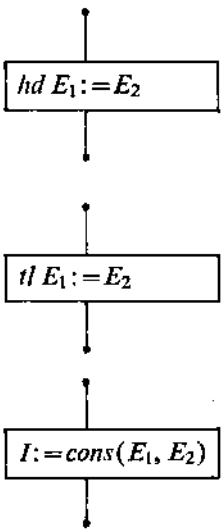
In the case of *cons* and atoms we assume a predicate *Atom*,

$Atom(u) \Leftrightarrow u \in A$, that is, $u \in U_a$ for some $a \in A$,

and (vii) yields

$*((u)\overset{\tau}{\rightarrow}(v_1, \ldots, v_n))$ and $u \neq v_i$, $i=1, \ldots, n$ and $Atom(u) \Rightarrow$
$(\exists\sigma)\tau = u \cdot \sigma$
and $\neg Atom(u) \Rightarrow (\exists\sigma)\tau = cons \cdot \sigma$.

In figure 11 we give the transformation sentences for this case. Their correctness is provable easily in just the same manner as those for lists. The extension to a general $\Omega$ should be obvious. The sentences for $I:=E$ and $E_1=E_2$ are unchanged and for the test $Atom(E)$ we just add the sentence $Atom(E)$ or its negation. Proposition 2 and the definition of $E$ go through unchanged.




Change set $=\{hd, *\}$
$*((E_1)\overset{cons}{\rightarrow}(u, v), \lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((E_1)\overset{cons}{\rightarrow}(E_2, v), \lambda_1, \ldots, \lambda_n)$

Change set $=\{tl, *\}$
$*((E_1)\overset{cons}{\rightarrow}(u, v), \lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((E_1)\overset{cons}{\rightarrow}(u, E_2), \lambda_1, \ldots, \lambda_n)$

Change set $=\{*, I\}$
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((I')\overset{cons}{\rightarrow}(E_1, E_2), \lambda_1, \ldots, \lambda_n)$

Figure 11. Transformation sentences for tree processing.

We now give a couple of examples of tree processing (or perhaps we should say 'expression processing'). Figure 12 gives a program in the form of a recursive subroutine for reversing a tree; for example $cons(a, cons(b, c))$ becomes $cons(cons(c, b), a)$. We have identified the one-tuple $(\tau)$ with $\tau$ to save on parentheses. The proof is by induction on the structure of $\tau$. Strictly we are doing induction on the free theory $T$. We define a tree reversing function recursively by

$rev(a) = a$ if $a$ is an atom
$rev(cons(\sigma, \tau)) = (cons(rev(\tau), rev(\sigma))$.

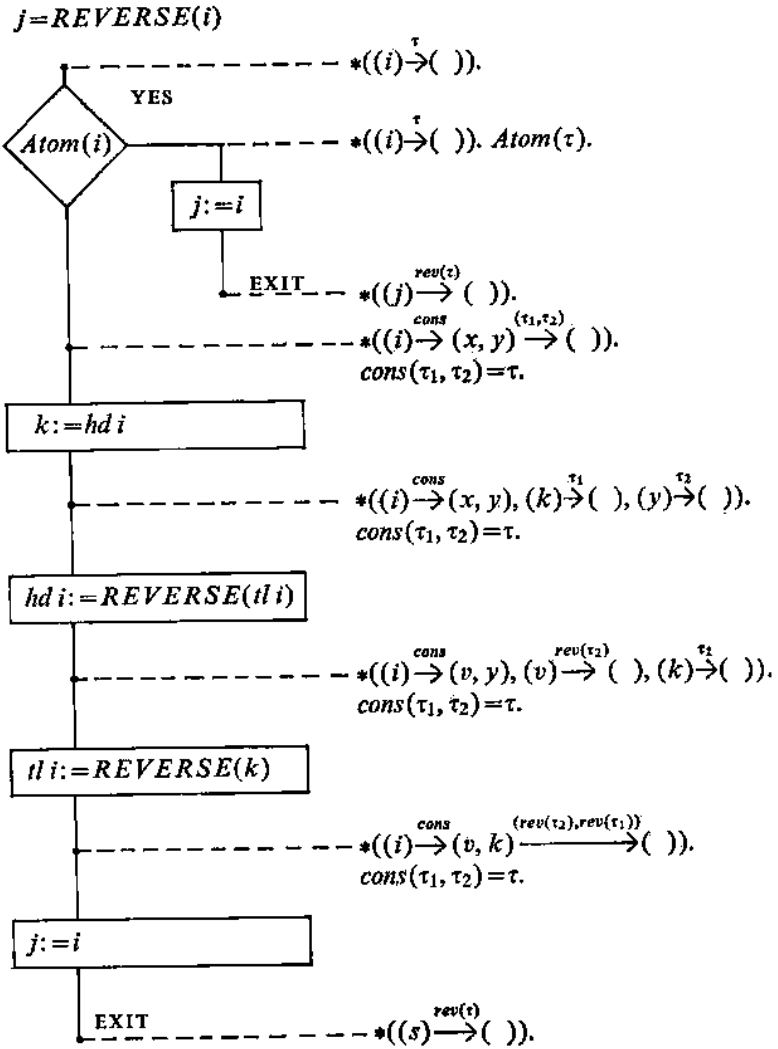



Figure 12. Reversing a tree.

$k = SUBST(i, a, j)$. Substitute $i$ for $a$ in $j$. $a$ is an atom.

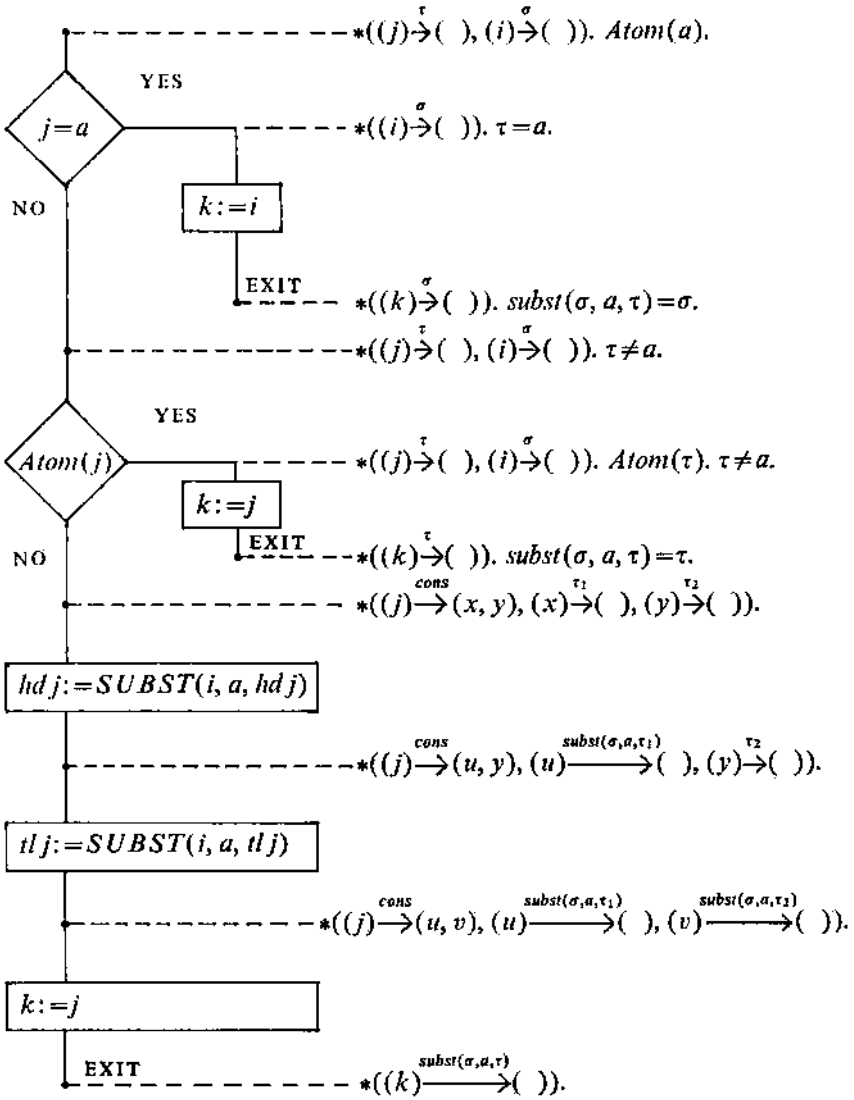



Figure 13. Substitution in a tree.

Figure 13 gives a recursive subroutine for substituting one tree into another wherever a given atom appears. Again the proof is by induction on the structure of $\tau$. We define substitution for morphisms of $T$ by

$subst(\sigma, a, a) = \sigma$

$subst(\sigma, a, b) = b$ if $b \neq a$ and $b$ is an atom

$subst(\sigma, a, cons(\tau_1, \tau_2)) = cons(subst(\sigma, a, \tau_1), subst(\sigma, a, \tau_2))$.

We should really have included some arbitrary extra morphisms $\lambda_i$ in the * expressions at entry and exit so as to carry through the induction, since the recursive calls act in the presence of other distinct morphisms; but this is clearly admissible and we have omitted it.

In our examples we have used the LISP case of *cons* and atoms, but even for LISP programs it might be useful to consider a more general $\Omega$. List structures are often used to represent expressions such as arithmetic expressions, using $cons(\text{'}PLUS\text{'}, cons(x, cons(y, nil)))$ for '$x+y$', similarly for '$x \times y$'. We can then allow $T$ to have binary operators + and × defining $(u) \overset{+}{\rightarrow} (v, w)$ if $hd(u) = \text{'}PLUS\text{'}$ and $hd(tl(u)) = v$ and $hd(tl^2(u)) = w$. This enables us to write the assertions intelligibly in terms of + and ×. In fact this representation of expressions corresponds to an injective functor between the category $T_{+,\times}$ and the category $T_{cons,A}$.

Free Theories as above seem appropriate where the trees have internal sharing only at known points. Data structures other than lists and trees remain to be investigated. More general categories of graphs with inputs and outputs might be worth considering. In general the choice of a suitable category analogous to $T$ would seem to depend on the subject matter of the computation, since we wish the vocabulary of the assertions to be meaningful to the programmer.

**Acknowledgements**

I would like to thank John Darlington and Gordon Plotkin for helpful discussions and Michael Gordon for pointing out some errors in a draft, also Pat Hayes and Erik Sandewall for helpful criticism during the Workshop discussion. I am grateful to Miss Eleanor Kerse for patient and expert typing. The work was carried out with the support of the Science Research Council. A considerable part of it was done with the aid of a Visiting Professorship at Syracuse University, for which I would like to express my appreciation.

## REFERENCES

Arbib, M.A. & Give'on, Y. (1968) Algebra automata I: parallel programming as a prolegomena to the categorical approach. *Information and Control*, **12**, 331–45. Algebra automata II: the categorical framework for dynamic analysis. *Information and Control*, **12**, 346–70. Academic Press.

Burstall, R.M. (1969) Proving properties of programs by structural induction. *Comput. Jl.*, **12**, 41–8.

Burstall, R.M. (1970) Formal description of program structure and semantics in first order logic. *Machine Intelligence 5*, pp. 79–98 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Eilenberg, S. & Wright, J.B. (1967) Automata in general algebras. *Information and Control*, **11**, 4, 452–70.
Floyd, R.W. (1967) Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19–32. Providence, Rhode Island: Amer. Math. Soc.
Good, D.I. (1970) Toward a man-machine system for proving program correctness. Ph.D. thesis. University of Wisconsin.
Hewitt, C. (1970) PLANNER: a language for manipulating models and proving theorems in a robot. *A.I. Memo. 168*. Project MAC. Cambridge, Mass.: MIT.
Hoare, C.A.R. (1971) Proof of a program: FIND. *Comm. Ass. comput. Mach.*, **14**, 39–45.
King, J.C. (1969) A program verifier. Ph.D. thesis. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.
Lawvere, F.W. (1963) Functorial semantics of algebraic theories. *Proc. Natl. Acad. Sci. U.S.A.*, **50**, 869–72.
London, R.L. (1970) Certification of Algorithm 245 – TREESORT 3. *Comm. Ass. comput. Mach.* **13**, 371–3.
MacLane, S. (1971) *Categories for the working mathematician*. Graduate Texts in Mathematics 5. Springer-Verlag.
McCarthy, J. (1963) A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pp. 33–70. (eds Braffort, P. & Hirschberg, D.). Amsterdam: North Holland Publishing Co.
McCarthy, J. & Painter, J.A. (1967) Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, pp. 33–41. Providence, Rhode Island: Amer. Math. Soc.
Wirth, N. & Hoare, C.A.R. (1966) A contribution to the development of ALGOL. *Comm. Ass. comput. Mach.*, **9**, 413–32.

## APPENDIX: FREE THEORIES

### Categories

By a category $C$ we mean a set $O$ of objects and a set $M$ of morphisms, together with a pair of functions $\partial_0, \partial_1 : M \to O$ (domain and co-domain), a partial operation of composition, $\cdot : M \to M$, and an identity operation $1 : O \to M$ such that

(i) $f \cdot g$ is defined iff $\partial_1 f = \partial_0 g$
(ii) If $f \cdot g$ and $g \cdot h$ are both defined then $(f \cdot g) \cdot h = f \cdot (g \cdot h)$
(iii) $\partial_0(1_a) = \partial_1(1_a) = a$
(iv) If $\partial_0 f = a$ and $\partial_1 f = b$ then $1_a \cdot f = f = f \cdot 1_b$

We write $f : a \to b$ as an abbreviation for $\partial_0 f = a$ and $\partial_1 f = b$ and write $C(a, b)$ for the set of all $f$ such that $f : a \to b$ in $C$. (For further development see, for example, MacLane 1971.)

### Functions over finite sets

For each integer $n \geqslant 0$, we write $[n]$ for the set $\{1, \ldots, n\}$.

We write $\varnothing$ for $[0]$ and $I$ for $[1]$ and notice that there are unique functions $\varnothing \to [n]$ and $[n] \to I$ for any $n$. A set $[n]$ and an integer $i \in [n]$ determine a unique function $I \to [n]$ which we will denote by $i$.

**The free theory on $\Omega$**

Let $\Omega=\{\Omega_n\}$, $n=0, 1, \ldots$ be a sequence of sets of operators.

We define the *free theory* on $\Omega$ inductively as the smallest category $T$ such that

(i) The objects of $T$ are the sets $[n]$, $n=0, 1, \ldots$

(ii) There is a function $d$ from the morphisms of $T$ to the non-negative integers. We call $d(f)$ the degree of the morphism $f$, and write $T_j([m], [n])$ for the set of all morphisms of degree $j$ from $m$ to $n$.

(iii) $T_0([n], [m])$ is the set of all functions from the set $[n]$ to the set $[m]$. Composition and identity are defined as usual for functions.

(iv) There is an operation of 'tupling' on morphisms from $I$ (as well as composition and identity). Thus for each $n$-tuple of morphisms $f_i: I\to[k]$, $i=1, \ldots, n$ there is a unique morphism $f$, written $\langle f_1, \ldots, f_n\rangle$ such that $i\cdot f=f_i$ for each $i=1, \ldots, n$. (Recall that $i: 1\to[n]$, is the function taking 1 to $i$ in $[n]$.) From the uniqueness we see that for any $f: [n]\to[k]$, $f=\langle 1\cdot f, \ldots, n\cdot f\rangle$.

The degree of $\langle f_1, \ldots, f_n\rangle$ is $d(f_1)+\ldots+d(f_n)$.

(v) $\Omega_n\subseteq T_1(I, [n])$.

(vi) If $\omega\in\Omega_m$ and $f\in T_j([m], [n])$ then $\omega\cdot f\in T_{1+j}(I, [n])$, and conversely if $g\in T_{1+j}(I, [n])$ for $j\geqslant 0$ then there is a unique $m$, a unique $\omega\in\Omega_m$ and a unique $f$ such that $g=\omega\cdot f$.

Now $T_{mn}$ in the body of the paper may be equated (to within an isomorphism) with $\bigcup_j T_j([m], [n])$ here.