VERIFICATION OF PROGRAMS WITH

DATA REFERENCING†

by

Michael R. Levy
Department of Computer Science
University of Waterloo
Waterloo, Ontario

CS-78-27
June 1978

Faculty

of

Mathematics

University of Waterloo
Waterloo, Ontario, Canada

VERIFICATION OF PROGRAMS WITH DATA REFERENCING[†]

by

Michael R. Levy
Department of Computer Science
University of Waterloo
Waterloo, Ontario

CS-78-27
June 1978

[†] This is a revised and expanded version of a paper presented at the 3rd Symposium on Programming, Paris, March 1978.

ABSTRACT: Several techniques have been proposed for proving properties of programs which manipulate data storage. These techniques all suffer the disadvantage of being based on "operational" models of the underlying type, rather than on an "abstract" denotational model. In this paper, the concepts of many-sorted algebras and algebraic congruences are used to describe an abstract data type called linked list. This type is equivalent to the LISP list data type, but it also allows operators equivalent to REPLACA and REPLACD, hence allowing completely general assignment. The abstract treatment of the type allows one directly to express concepts such as sharing and also to reason directly about structures manipulated by programs. Axioms are presented for the type and a simple proof rule which is general enough for any assignment statement is given. Some properties of the type are discussed. Finally, a program is presented together with its assertions to illustrate correctness proofs of programs using the type.

1. INTRODUCTION

Several techniques have been proposed for proving properties of programs which manipulate data storage. (See for example Burstall 1972 and Kowaltowski 1976). The issue is to develop a proof system for programs which operate on complex data structures possibly with shared components. The semantics on which these methods are based are usually of the Scott-Strachey type (Scott and Strachey 1972), or a similar "machine" model which assumes explicitly or implicitly the existence of "nodes" and node identifiers or locations with associated values. (In real machine terms these would be words and addresses.) These concepts are, however, not relevant to the semantics of programs manipulating referenced data, because they often involve extraneous details, and proofs in systems using such "operational" semantics are likely to be difficult.

We outline in this paper a semantics of a general data type (linked list) which allows sharing and circularity that is not based on an operational or abstract-machine semantics. We show that it is possible to describe concepts such as sharing and "in situ" manipulation without recourse to operational devices such as addresses. A simple proof rule for the assignment operator is presented; this proof rule can be used with any standard verification technique such as the invariant assertion method (Hoare 1969). Furthermore, the model can be described using well-known mathematical concepts or using the notion of Algebraic Specifications (ADJ 1977b, Guttag 1977). (In fact, we use many-sorted algebras, but only a superficial knowledge of this subject is required to read the paper. The required definitions and theorems used are stated in section 2.) Two important concepts from the theory of many-sorted algebras are used - firstly, initiality is used to describe

conveniently the syntax of the languages under consideration, and secondly, the notion of quotient is used for an "abstract" treatment of the type described in the paper. Neither of these concepts should provide difficulty to readers with an understanding of the more general mathematical notions of syntax and congruences. The two papers by Goguen et al provide an excellent discussion of these topics (ADJ 1977a, 1977b).

The difficulty with verifying programs which manipulate structures with sharing arises mostly from the fact that a single assignment statement can affect the values of several variables. For example, consider the sequence of statements

1. $x \leftarrow \text{cons}(a,\text{cons}(b,c))$
2. $y \leftarrow t\ell\ x$
3. $t\ell\ y \leftarrow z$

where $\leftarrow$ is the (usual) assignment operator, and $t\ell$ is a function which returns the second component of a cell. (We say

$$t\ell(\text{cons}(c_1,c_2)) = c_2;$$

an intuitive discussion of the type is presented below.) $t\ell$ corresponds to CDR in LISP (McCarthy et al 1962), and the assignment $t\ell\ y \leftarrow z$ is equivalent to REPLACD(y,z) in LISP. Intuitively, the second assignment statement causes y to share the $t\ell$ of x. Hence the third assignment statement will change the value of x as well as the value of y. If we write down an assertion as the post condition of statement 3, say

Q: $x = \text{cons}(a,\text{cons}(b,\text{cons}(d,e))) \ \&\ y = \text{cons}(b,\text{cons}(d,e)) \ \&\ z = \text{cons}(d,e)$,

then the sharing between x, y and z cannot be explicitly determined. For

example, the following three assignments also satisfy the above assertion

1. x ← cons(a,cons(b,cons(d,e)))
2. y ← cons(b,cons(d,e))
3. z ← cons(d,e)

but in this case, there is no sharing.

In the above assertion we assumed that the values of the variables x, y and z were the associated lists cons(a,cons(b,cons(d,e))), cons(b,cons(d,e)) and cons(d,e) respectively. This is consistent with assertions about "simple" variables, where we may write

$$x = 3 \ \& \ y > 25$$

as an assertion. 3 is the value of x, and the value of y is greater than 25. There are two difficulties in treating variables and values of list structures in this way. Firstly, an assignment statement of the form

$$t\ell\ x \leftarrow \ell$$

can introduce circularities into the list structure, for example

$$x \leftarrow \text{cons}(a,b)$$
$$t\ell\ x \leftarrow x$$

after which the value of x would be recursive in x:

$$x = \text{cons}(a,x)$$

Is this value different from

$$x = \text{cons}(a,\text{cons}(a,x))?$$

The second difficulty is that the contents of a "cell" are not sufficient to identify that cell uniquely: hence the two possible interpretations of the assertion Q stated above. It is possible to formulate proof rules for the assignment statement, assuming some operational model of lists. (See for example Oppen and Cook 1975.) These rules tend to be extremely complex, however, and hence difficult to use in proofs. The approach taken here is based on the observation that an assignment statement (such as

$$t\ell\ y \leftarrow z$$

above) changes not only the value of y, but in fact the entire list structure with which y is associated. This view is similar to the view that an array assignment such as

$$A[i] \leftarrow e$$

changes the entire array A (See Reynolds 1978).

Hence suppose that σ is the name of a data structure. We view an assignment

$$px \leftarrow e$$

(which implicitly refers to a particular structure σ) as in fact being the assignment

$$\sigma' \leftarrow U(\sigma,px,e)$$

where $U(\sigma,px,e)$ is a function which changes σ by changing px to e, and $p \in \{hd,t\ell\}^*$, that is px is an expression in terms of hd and $t\ell$. Hence the proof rule we need is

$$Q^{\sigma}_{U(\sigma,px,e)}\ \{px \leftarrow e\}\ Q$$

where $Q^{\sigma}_{U(\sigma,px,e)}$ is Q with each occurrence of σ replaced by $U(\sigma,px,e)$. (Compare this rule with the "simple" assignment rule in Hoare 1969 and the array rule in Reynolds 1978.)

For the sake of notation, we will write σ' for $U(\sigma,px,e)$ provided px and e can be determined from the context of the assertion. Hence the proof rule will be written

$$Q^{\sigma}_{\sigma'} \{ px \leftarrow e \} Q$$

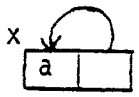
where σ' depends on the particular assignment.

In order for this simple proof rule to work, it is necessary to write assertions about some structure σ rather than components of the structure. This, together with the treatment of referencing discussed below, allows properties of a structure to be simply expressed. It also allows one structure to be compared to another: for example the final "value" of a structure can be compared with the initial structure by treating the initial structure as a constant in the assertion. These ideas are illustrated in the final section.

Finally, in order to be able to use the proof rule presented above, the concepts of cell and reference must be carefully examined. A cell is identified by a reference and its contents are denoted by a description. References must have the property that they uniquely identify a cell, while two distinct cells may have the same description. Note that the description of a particular cell depends on the particular structure (or state) in which the cell occurs. The process of describing a particular cell (that is, giving a description of a cell) is called dereferencing. References correspond to the L-values of Scott and Strachey 1972 and descriptions to the

R-values. However, a function could have a reference as a value, and we have thus chosen the words reference and description in order not to overwork the word value. In Algol 68, descriptions would be called values. (See Barron 1977.)

In general, assertions about programs will be written in terms of references rather than values. As will be seen below, this has the advantage of allowing the proof rule discussed previously to be used for assignment but also allows one to state concepts such as "in place" and "circularity" in a natural and concise way. For example, the simple circularity



discussed above can be described by the predicate

$$x(\sigma) = t\ell\ x(\sigma) \quad \& \quad \downarrow(hd\ x(\sigma),\sigma) = a$$

where $\downarrow$ is used to denote the dereferencing operator.

Finally, it is necessary to be able to manipulate equations involving references. For example, when using the proof rule for the assignment statement

$$t\ell\ y \leftarrow z$$

if the post-condition is say

$$Q: \quad t\ell\ t\ell\ y(\sigma) = x(\sigma)$$

then the precondition would be

$$Q^{\sigma}_{U(\sigma,t\ell\ y,z)}: \quad t\ell\ t\ell\ y(U(\sigma,t\ell\ y,z)) = x(U(\sigma,t\ell\ y,z)).$$

Now in order for the proof rule to be useful in a program proof, it must be possible to eliminate the term $U(\sigma, t\ell\ y,z)$ and have the assertion $Q^{\sigma}_{U(\sigma,t\ell\ y,z)}$ reduced to an equivalent assertion P which is expressed in terms of σ only. (In the case of the array proof rule in Reynolds, the term $U(A,i,e)$ will be eliminated so that only A is referred to in assertions.)

In the above example, it may be reasonable to conclude that

$$t\ell\ t\ell\ y(U(\sigma,t\ell\ y,z)) = t\ell\ z(\sigma) \qquad \text{(since } t\ell\ y \text{ is changed to } z\text{)}$$

and that

$$x(U(\sigma,t\ell\ y,z)) = x(\sigma) \qquad \text{(since } x \text{ cannot have been changed),}$$

and hence $Q^{\sigma}_{U(\sigma,t\ell\ y,z)}$ is clearly implied by the precondition

$$P:\ t\ell\ z(\sigma) = x(\sigma)\ .$$

(Intuitively, if tl z and x refer to the same cell in a structure and the $t\ell$ of y is made to refer to z, then after the update the $t\ell$ of the $t\ell$ of y must refer to that cell.)

In order to provide the appropriate "calculus" for manipulating equations using references, we define the type linked list using the view that data types are (many-sorted) algebras. (This view is discussed in detail by ADJ 1977b.) The important property of this view is that a type so defined (as an algebra) can be characterized as a quotient of the word algebra (defined below). This gives us two useful properties:

1. The equational specification (axioms) used to define the type can be used to make the kind of reduction referred to in the above

example: two terms are equivalent in the quotient if one can be derived from the other using the axioms.

2. The quotient is defined by a congruence: if two terms are equivalent, then they behave identically with respect to the operators of the type. Thus one has referential transparency at least with respect to the operators. For example if

$$r_1 = r_2$$

where r_1 and r_2 are references, then

$$hd(r_1)(\sigma) = hd(r_2)(\sigma).$$

Because of the particular way in which the type is defined here, it is possible to give a formal definition of the concept of a cell and also to relate cells in a structure σ to cells in a structure σ_0 if σ was derived from σ_0 by a series of assignments. The next section gives the necessary mathematical preliminaries. More details can be found in ADJ(1977b). In section 3 the type linked list is defined by giving the definition of the type as an algebra with the associated axioms. The form of these axioms makes it possible to use them directly in verification to make reductions of the form discussed above. In section 4, a simple program from Burstall (1972) is presented and a sample of the correctness proof is given. Section 5 contains the conclusions.

2. PRELIMINARY DEFINITIONS AND RESULTS

A data type is viewed as a many-sorted algebra. Discussion of data types as many-sorted algebras can be found in ADJ (1977b), Guttag (1977) and Levy (1977). An algebra of one sort is roughly-speaking a set of objects and a family of operators on the set. The set is called the carrier of the algebra. Many-sorted algebras extend this notion by allowing the carrier of the algebra to consist of many disjoint sets. Each of these sets is said to have a sort. The operators are sorted or typed, but must be closed with respect to the carrier. For example, if A,B,C are three sets in the carrier of an algebra, then

$$+ : A \times B \to C$$

could be an operator of type $<ab,c>$, arity ab and sort c, where a,b and c are distinguished names (the sorts) of A,B and C respectively. A data type is then a many-sorted algebra, while a data structure is an element of the carrier of a data type.

Let S be a set whose elements are called sorts. An S-sorted operator domain Σ is a family $\Sigma_{\omega,s}$ of sets of symbols, for $s \in S$ and $\omega \in S^*$ where S^* is the free monoid on S. $\Sigma_{\omega,s}$ is the set of operator symbols of type $<\omega,s>$, arity ω and sort s.

A Σ-algebra A consists of a family $<A_s>_{s \in S}$ of sets called the carrier of A, and for each $<\omega,s> \in S^* \times S$ and each $\sigma \in \Sigma_{\omega,s}$, a function

$$\sigma_A: A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \to A_s$$

(where $\omega = s_1 s_2 \ldots s_n$) called the operation of A named by σ.

Here $\langle x_s \rangle_{s \in S}$ denotes a family of objects x_s indexed by s, such that there is exactly one object x_s for each $s \in S$. The subscript $s \in S$ will be omitted when the index set S can be determined from the context. For $\sigma \in \Sigma_{\lambda,s}$ where λ is the empty string, $\sigma_A \in A_s$ (also written $\sigma_A : \to A_s$). These operators are called constants of A of sort s.

If $\omega = s_1 s_2 \ldots s_n$, then let A^ω denote $A_{s_1} \times \ldots \times A_{s_n}$.

If A and A' are both Σ-algebras, then a Σ-homomorphism $h: A \to A'$ is a family of function

$$\langle h_s : A_s \to A'_s \rangle_{s \in S}$$

such that if $\sigma \in \Sigma_{\omega,s}$ and $\langle a_1,\ldots,a_n \rangle \in A^\omega$ then

$$h_s(\sigma_A(a_1,\ldots,a_n)) = \sigma_{A'}(h_{s_1}(a_1),\ldots,h_{s_n}(a_n)).$$

A Σ-algebra A in a class $\underline{\underline{C}}$ of Σ-algebras is said to be initial in $\underline{\underline{C}}$ iff for every B in $\underline{\underline{C}}$ there exists a unique homomorphism

$$h: A \to B.$$

THEOREM 1 The class of all Σ-algebras has an initial algebra called T_Σ. □

(T_Σ is sometimes called the free Σ-algebra.) This theorem, as well as the others given in this section, is proved in (ADJ 1977b). T_Σ can be viewed intuitively as the algebra of well formed expressions over Σ.

Variables can be included in terms in T_Σ in the following way. Let

$$X_s = \{x_s^{(n)} \mid n \in N\}$$

where N is the set of natural numbers. The elements

$$x_s^{(i)} \in X_s$$

are symbols called variables of sort s. Suppose Σ is an S-sorted operator domain. Let

$$X = \bigcup_{s \in S} X_s$$

where X_s is a family of variables of sort s for each $s \in S$. We say that X is an S-indexed family of variables.

Then let $\Sigma(X)$ be the s-sorted operator domain defined as follows:

$$\Sigma(X)_{\lambda,s} = \Sigma_{\lambda,s} \cup X_s$$

$$\Sigma(X)_{\omega,s} = \Sigma_{\omega,s} \quad \text{for} \quad \omega \neq \lambda.$$

Thus variables are being treated as nullaries or constants. Clearly $T_{\Sigma(X)}$ is an initial $\Sigma(X)$-algebra. We define $T_\Sigma(X)$ as the algebra with the same carrier as $T_{\Sigma(X)}$, but with operations Σ. We say that $T_\Sigma(X)$ is the Σ-algebra freely generated by X.

For any S-sorted Σ-algebra A and an S-indexed family X of variables, if

$$\theta: X \to A$$

denotes a family of functions

$$< \theta_s: X_s \to A_s >$$

then θ is called an interpretation or assignment of values of sort s in A to variables of sort s in X.

THEOREM 2 Let A be a Σ-algebra and $\theta: X \to A$ an assignment. Then there exists a unique homomorphism

$$\overline{\theta}: T_\Sigma(X) \to A$$

that extends θ in the sense that $\overline{\theta}_s(x) = \theta_s(x)$ for all $s \in S$ and $x \in X$. □

A Σ-equation is a pair $e = \langle L,R \rangle$ where $L,R \in T_{\Sigma(X),s}$ (the carrier of $T_\Sigma(X)$ of sort s). A Σ-algebra A satisfies e if

$$\overline{\theta}(L) = \overline{\theta}(R)$$

for all assignments $\theta: X \to A$. If ε is a set of Σ-equations, then A satisfies ε iff A satisfies each $e \in \varepsilon$. Thus a set of equations ε can be viewed as a set of axioms whose free variables are implicitly universally quantified. The class of Σ-algebras which satisfy ε is denoted $\underline{Alg}_{\Sigma,\varepsilon}$.

An equational specification is a triple $\langle s,\Sigma,\varepsilon \rangle$ where Σ is an S-sorted operator domain and ε is a set of Σ-equations (called the type axioms).

Let $w = s_1 s_2 \ldots s_n$ and $a_i, a_i' \in A_{s_i}$ for $1 \le i \le n$. Then a Σ-congruence $\equiv$ on a Σ-algebra A is a family $\langle \equiv_s \rangle_{s \in S}$ of equivalence relations $\equiv_s$ on A_s such that if $\sigma \in \Sigma_{\omega,s}$ and if $a_i \equiv_{s_i} a_i'$ for $1 \le i \le n$, then

$$\sigma_A(a_1,\ldots,a_n) \equiv_s \sigma_A(a_1',\ldots,a_n').$$

If A is a Σ-algebra and $\equiv$ is a Σ-congruence on A, let $A/\equiv = \{A_s/\equiv_s\}_{s \in S}$ be the set of $\equiv_s$-equivalence classes of A_s. For $a \in A_s$ let $[a]_s$ denote the $\equiv_s$-class containing a. It is possible to make $A/\equiv$ into a Σ-algebra by defining the operations $\sigma_{A/\equiv}$ as follows:

(i) If $\sigma \in \Sigma_{\lambda,s}$ then $\sigma_{A/\equiv} = [\sigma_A]_s$

(ii) If $\sigma \in \Sigma_{s_1 \ldots s_n,s}$ and $[a_i]_{s_i} \in A_{s_i}/\equiv_{s_i}$ for $1 \le i \le n$

then

$$\sigma_{A/\equiv}([a_1]_{s_i},\ldots,[a_n]_{s_i}) = [\sigma_A(a_1,\ldots,a_n)]_s$$

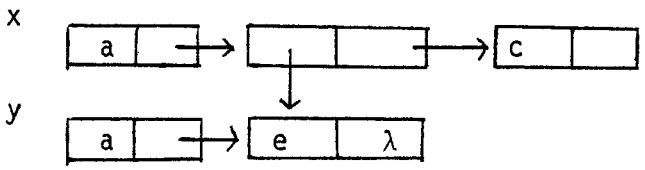
Then it can be shown that $A/\equiv$ is a Σ-algebra called the quotient of A by $\equiv$. (The property of $\equiv$ being a congruence ensures that $\sigma_{A/\equiv}$ is well defined.)

A set of Σ-equations $\varepsilon = \{ < L,R> \mid L,R \in T_\Sigma(X)\}$ generates a binary relation $R \subseteq A \times A$ on any algebra A. This relation is the set of all pairs $\{ < \overline{\theta}(L),\overline{\theta}(R)> \mid \theta$ is an assignment$\}$. (Intuitively, consider all pairs with variables in L and R replaced by terms in T_Σ and the resulting expressions then being evaluated in A_Σ.) Now any relation R on algebra A generates a congruence q on A which is the least congruence on A containing R.

THEOREM 3 If ε is a set of Σ-equations generating a congruence q on T_Σ, then T_Σ/q is initial in $\underline{\underline{Alg}}_{\Sigma,\varepsilon}$. □

3. THE TYPE LINKED LISTS

The type to be described has essentially the same semantics as the LISP type (McCarthy et al 1962), but uses Algol 60 syntax (as for example in Burstall 1972). Roughly speaking an element of the type is called a list structure (or structure), and it consists of a finite set of cells or nodes. Each cell has two components called the head and tail of the cell. The head and tail of a cell each contain a reference to another cell or an element called an atom. For example, we can represent a typical list using a "box and arrow" diagram:

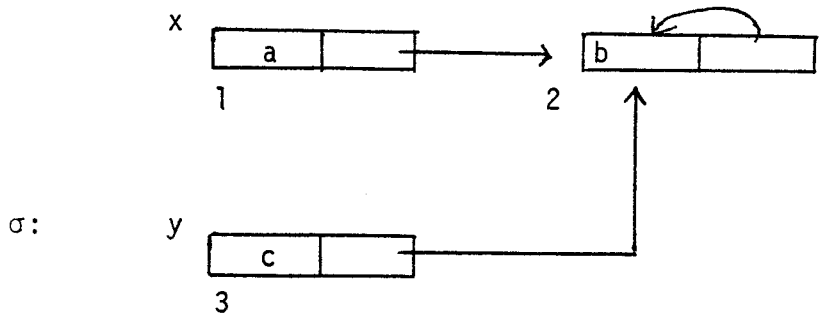


a,c,e and λ are all atoms, with λ being a distinguished element called null.

The operators of the type include the following three operators: hd and $t\ell$ which take as argument a cell and return the head and tail of the cell respectively, and cons which takes two arguments (atoms or cells) and returns as its value the description of a cell containing in its head and tail its two arguments. There are also three assignment operators in the type. Simple assignment, $\leftarrow(x,c,\sigma)$ associates a variable x with a cell c in a given structure σ. More general assignment operators change the head or tail of a cell. These assignments are $\leftarrow_{hd}(c_1,c_2,\sigma)$ (or $\leftarrow_{t\ell}(c_1,c_2,\sigma)$) and the intended semantics is that the head (or tail) of cell c_1 is changed to refer to c_2. hd and $t\ell$ correspond to CAR and CDR in LISP, while $\leftarrow$ corresponds to CSET. $\leftarrow_{hd}(c_1,c_2,\sigma)$ corresponds

to REPLACA(c_1,c_2) and $\leftarrow_{t\ell}(c_1,c_2,\sigma)$ to REPLACD(c_1,c_2). Note that REPLACA and REPLACD were never formally defined in LISP because of the difficulty of dealing with sharing in the LISP formalism.

The simplest form of a reference is denoted by the name of a variable which refers in some structure to a cell. More complex references can be built by examining the head or tail of a referenced cell. Consider for example the structure



Box 1 is referenced by x, box 3 by y and box 2 by the tail of the cell referenced by x, the tail of the cell referenced by y as well as by the tail of itself. Letting $x(\sigma)$ and $y(\sigma)$ denote the cells referenced in σ by x and y respectively, then box 2 is referenced in σ by $t\ell(x(\sigma))(\sigma)$ and $t\ell(y(\sigma))(\sigma)$. Note that the cell referenced by x in σ (say) may have different contents at different "times". Hence the value of the function $t\ell$ is itself a function of the same structure. If we regard $t\ell(r)$ as a function, which given some structure σ, returns the tail of the cell referenced by r in structure σ, then we can associate with each cell in the above diagrams the references:

Cell 1. $x(\sigma)$

Cell 3. $y(\sigma)$

Cell 2. $t\ell(x(\sigma))(\sigma)$; $t\ell(y(\sigma))(\sigma)$; $t\ell(t\ell(x(\sigma))(\sigma))(\sigma)$; $t\ell(t\ell(y(\sigma))(\sigma))(\sigma)$; ...

Suppose R is the set of all well-formed expressions of the above form over hd, tℓ, x and y and σ. Then the structure σ partitions R into classes such that the elements of a given class all reference the same cell. (There is also a class of "undefined" references.)

Suppose that

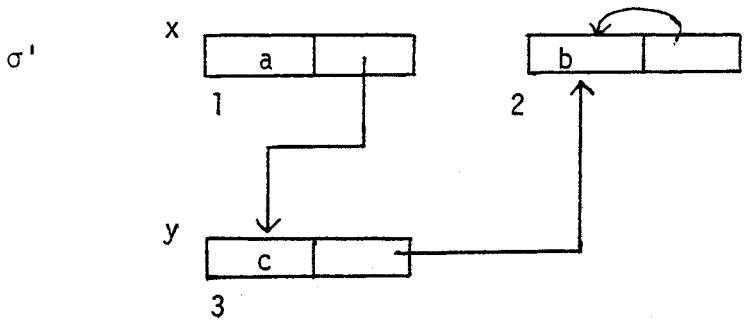
$$\sigma_0, \sigma_1, \ldots, \sigma_n$$

is a sequence of structures where σ_i is obtained from σ_{i-1}, $i > 0$, by an update. Then references in σ_j can be related to references in σ_k, $0 \leq j, k \leq n$, by relating cells in the updated structure to cells in the original structure. Conceptually, the new set of cells is obtained from the old by changing one of the cells in the old set of cells, or by associating a variable with a different cell. We can thus identify particular cells in the old and the new structures.

For example, suppose the above structure is updated by the assignment

$$\leftarrow_{t\ell}(x, y, \sigma),$$

giving σ'. The updated structure is (pictorially)



In σ' the partition is

Cell 1. $x(\sigma')$

Cell 3. $y(\sigma')$, $t\ell(x(\sigma'))(\sigma')$

Cell 2. $t\ell(y(\sigma'))(\sigma')$, $t\ell(t\ell(x(\sigma'))(\sigma'))(\sigma')$ ...

as well as a number of expressions which are regarded as being undefined. We can thus relate references in σ' to σ: Let R' be the set R together with well-formed expressions in hd, $t\ell$, x, y and σ'. Then partition R' so that all elements in a class of the partition reference the same cell. For example, $x(\sigma)$ and $x(\sigma')$ will be in the same class; $t\ell(y(\sigma'))(\sigma')$ and $t\ell(x(\sigma))(\sigma)$ will be in the same class. It is also possible to "mix" structures. For example, $x(\sigma')$ refers to box 1 in σ and σ', so we can consider $t\ell(x(\sigma'))(\sigma)$ (which refers to box 3). If σ and σ' are unrelated, that is neither is derived by applying zero or more updates to the other or they are not obtained by updating the same structure, then expressions involving both σ and σ' are considered to be undefined.

Descriptions are denoted by expressions of the form

$$cons(d_1,d_2)$$

where d_1,d_2 are atoms, references or descriptions. Descriptions do not themselves identify cells, but expressions involving descriptions may. Consider for example, the expression

$$hd(cons(r_1,r_2))(\sigma)$$

where r_1 is a reference in σ. The subexpression

$$cons(r_1,r_2)$$

is the description of some cell with the reference r_1 in its head. Thus

$$hd(cons(r_1,r_2))(\sigma) = r_1$$

is a reference and does identify a cell.

Finally we notice that references behave in the following "algebraic" way: if r_1 and r_2 reference the same cell, then $hd(r_1)(\sigma)$ and $hd(r_2)(\sigma)$ will either be equal as atoms or will both reference the same cell. Similarly, $t\ell(r_1)(\sigma)$ and $t\ell(r_2)(\sigma)$ are equal as atoms or reference the same cell. In addition, if c_1 references the same cell as c_2 or c_1 and c_2 are the same descriptions (we write $c_1 = c_2$) and $c_1' = c_2'$, then

$$cons(c_1,c_1') = cons(c_2,c_2')$$

These informal ideas can all be formalized by defining a congruence over well-formed expressions in the operators of the type. The congruence captures both the notion of equivalence (of references that refer to the same cell and of descriptions which are the same) and the notion of the operators of the type preserving this equivalence. The congruence is described by a set of equations of the type, called type axioms.

The carrier of the type linked list will consist of four sets: A set of variables called <u>program variables</u> of sort V, a set of sort Cell, a set of sort Ref and a set of sort Struct. The operator domain Σ is:

ϕ:	$\rightarrow$ Struct	
hd,$t\ell$:	Ref x Struct $\rightarrow$ Ref	
cons :	Ref x Ref $\rightarrow$ Cell	
x:	Struct $\rightarrow$ Ref	for each $x \in V$
$\bar{x}$:	$\rightarrow$ V	for each $x \in V$
a:	$\rightarrow$ Cell	for each $a \in$ Atom

$\leftarrow$: V x Ref x Struct → Struct

$\leftarrow_{hd}$: Ref x Ref x Struct → Struct

$\leftarrow_{t\ell}$: Ref x Ref x Struct → Struct

_: Cell → Ref

↓: Ref x Struct → Cell

isref: Ref → Bool

We have also assumed that the sort Bool with operators

$$\underline{\text{true}}, \underline{\text{false}}: \rightarrow \text{Bool}$$

$$\text{if then else: Bool x } t \text{ x } t \rightarrow t \qquad \text{for each sort } t$$

is implicitly part of the type. The axioms for these operators are the usual "if then else" axioms, namely

$$\text{if then else } (\underline{\text{true}}, t_1, t_2) = t_1$$

$$\text{if then else } (\underline{\text{false}}, t_1, t_2) = t_2.$$

The notation

$$\_: \text{Cell} \rightarrow \text{Ref}$$

denotes the fact that each element in the carrier of sort Cell is in the carrier of sort Ref. It may also be viewed as a conversion operator which is omitted syntactically.

This definition of Σ is sufficient to determine the algebra T_Σ whose elements can be considered to be the well formed expressions over Σ. If we denote the set in the carrier of T_Σ of sort Cell by Cell, then cons(a,cons(b,cons(c,d))) $\in$ Cell, assuming that a, b, c and d $\in$ Atom,

for some distinguished set of elements called atoms. Similarly, if we denote the set in the carrier of T_Σ of sort Struct by Struct, then

$$\leftarrow (x,cons(a,cons(b,cons(c,d))),\phi) \in \text{Struct}.$$

The most significant aspect of the above definition of Σ is in the treatment of references. If x is a variable, $x \in V$, then

$$x: \text{Struct} \rightarrow \text{Ref}$$

says that the cell referenced by x depends on some particular structure. Hence the cell referenced by x in two different structures is not necessarily the same. Similarly, hd and $t\ell$ depend both on some reference r and on some structure σ. We will write $hd(r)(\sigma)$ (or $t\ell(r)(\sigma)$) to denote $hd(r,\sigma)$ (or $t\ell(r,\sigma)$.) Hence $hd(t\ell(x(\sigma))(\sigma))(\sigma) \in \text{Ref}$ for some $\sigma \in$ Struct. The elements of Ref which are of the form $\delta_1(\delta_2(\ldots(\delta_n(x(\sigma_{n+1}))(\sigma_n)\ldots)(\sigma_1)$ where $\delta_i \in \{hd,t\ell\}$, $\sigma_i \in$ Struct and $1 \le i \le n$, will be called <u>references</u>. If $\sigma_i = \sigma$ for each i, $1 \le i \le n+1$, the element will be called a <u>reference in σ</u>.

The quotient defined by the set of axioms to be presented will have the property that it groups together all references which have equal value. Also note that intuitively, a structure carries with it its "history". For example, the structure pictured in the example above (page 15) may have been derived from

$$\sigma = \leftarrow (t\ell t\ell\ x, t\ell\ x, \leftarrow(y,cons(c,t\ell\ x), \leftarrow(x,cons(a,cons(b,\lambda)),\phi))).$$

The "box and arrow" diagram in fact does not characterize a structure in the sense intended here, since there are different sequences of assignments that will "yield" the same diagram.

Remark on Notation

(i) Expressions involving the assignment operators $\leftarrow_{hd}$ and $\leftarrow_{t\ell}$ will be written using the operator $\leftarrow$ with hd or $t\ell$ to the left of the first argument to the operator. Thus we will write

$$\leftarrow(\delta r_1, r_2, \sigma)$$

to denote

$$\leftarrow_{\delta}(r_1, r_2, \sigma)$$

where $\delta \in \{hd, t\ell\}$.

(ii) If every structure in a reference is the same, the structure will only be written once. Hence for

$$t\ell\ (hd(t\ell(y(\sigma))(\sigma))(\sigma))(\sigma)$$

we will simply write

$$t\ell\ hd\ t\ell\ y(\sigma).$$

(iii) In an assignment of the form

$$\leftarrow(x, r_2, \sigma) \quad \text{or} \quad \leftarrow(\delta r_1, r_2, \sigma),$$

if all structures occurring in the expressions r_1 and r_2 are σ, then σ is omitted from r_1 and r_2.
For example, we write

$$\leftarrow(t\ell\ x,\ cons(t\ell\ y,\ z), \sigma)$$

to denote

$$\leftarrow(t\ell\ x(\sigma),\ cons(t\ell\ (y(\sigma))(\sigma),\ z(\sigma)), \sigma).$$

Before presenting the type axioms, we discuss the intended semantics of the three assignment operators $\leftarrow$, $\leftarrow_{hd}$, and $\leftarrow_{t\ell}$.

(i) $\leftarrow(x,r,\sigma)$ updates σ to a structure σ' which is identical to σ except that $x(\sigma')$ is a reference to a different cell. If r is a reference, then $x(\sigma')$ references the cell referenced by r. If r is a description, then $x(\sigma')$ references a "new" cell which has a description equal to r. We call this type of assignment <u>simple</u> assignment.

(ii) $\leftarrow(\delta r_1, r_2, \sigma)$ updates σ to a new structure σ' which is identical to σ except that the contents of cell r_1 are changed. If δ = hd, then the head of r_1 is changed to reference r_2. If $\delta = t\ell$, then the tail of r_1 is changed to reference r_2. We do not allow r_2 in this case to be a description, but this type of assignment can always be achieved at the expense of an extra variable since

$$\leftarrow(\delta r_1, \text{cons}(r_2,r_3),\sigma)$$

(which is not allowed) is equivalent to

$$\leftarrow(\delta r_1, z, \leftarrow(z, \text{cons}(r_2,r_3),\sigma)).$$

<u>Definition: The Type Axioms</u>

Let $r, r_1, r_2, r_3, r_4 \in \text{Ref}$, $\sigma, \sigma_1, \sigma_2 \in \text{Struct}$ and $p,q \in \{\text{hd},\text{tl}\}^*$. Then

1. $\text{hd}\ (\text{cons}(r_1,r_2))(\sigma) = r_1$
2. $t\ell\ (\text{cons}(r_1,r_2))(\sigma) = r_2$
3. $x\ (\leftarrow_{hd}\ (r_1,r_2,\sigma)) = x(\sigma)$
4. $x\ (\leftarrow_{t\ell}\ (r_1,r_2,\sigma)) = x(\sigma)$
5. $$qx\ (\leftarrow(y,r,\sigma)) = \begin{cases} qr(\sigma) & \text{if } x == y \ \&\ \text{isref}(qr(\sigma)) \\ qx(\sigma) & \text{if } \sim(x == y) \end{cases}$$

6. $hd(r)(\leftarrow_{tl}(r_1,r_2,\sigma)) = hd(r)(\sigma)$

7. $tl(r)(\leftarrow_{hd}(r_1,r_2,\sigma)) = tl(r)(\sigma)$

8. $$hd(r)(\leftarrow_{hd}(r_1,r_2,\sigma)) = \begin{cases} r_2 & \text{if } \approx(r,r_1) \\ hd(r)(\sigma) & \text{otherwise} \end{cases}$$

9. $$tl(r)(\leftarrow_{tl}(r_1,r_2,\sigma)) = \begin{cases} r_2 & \text{if } \approx(r,r_1) \\ tl(r)(\sigma) & \text{otherwise} \end{cases}$$

10. $\approx(px(\sigma_1), qy(\sigma_2)) = (px == qy \;\&\; \sigma_1 == \sigma_2)$
if $\sim$ isref$(px(\sigma_1))$ & $\sim$ isref$(qy(\sigma_2))$

11. $\approx(cons(r_1,r_2), cons(r_3,r_4)) = false$

12. $\approx(a,b) = false$ $\quad a,b \in Atom$

13. $$isref(r) = \begin{cases} \underline{false} & \text{if } r \in Atom \text{ or } r = cons(r_1,r_2) \\ \underline{true} & \text{otherwise} \end{cases}$$

The axioms may be read intuitively as follows

1,2. These are the usual list axioms.

3,4. The cell referenced by a variable is not changed by the assignments $\leftarrow_{hd}$ and $\leftarrow_{tl}$.

5. A simple assignment to y will affect any reference expressed in terms of y. However, care must be taken not to identify a reference with a description. For example, consider the structure σ' obtained by

$$\sigma' = \leftarrow(y, cons(r,b),\sigma).$$

If r is not a description, say

$r = x(\sigma)$, then we would expect that

$hd\ x(\sigma') = hd(hd\ (cons(x(\sigma),b))(\sigma)$

$= x(\sigma).$

However, if r is a description, say

$r = cons(a,b)$, we would not want

$$hd\ x(\sigma') = hd(cons(cons(a,b),b))(\sigma)$$
$$= cons(a,b)$$

because the description $cons(a,b)$ does not uniquely identify the cell referenced by $hd\ x(\sigma')$. This is why the term $isref(qr(\sigma))$ appears on the right hand side of axiom 5.

6,7. $\leftarrow_{tl}$ never affects the head of a cell and $\leftarrow_{hd}$ never affects the tail of a cell.

8,9. These axioms describe the assignments $\leftarrow_{hd}$ and $\leftarrow_{tl}$. The symbol $\approx$ may be interpreted as equality of references. Thus the head of a cell r is affected by the assignment $\leftarrow_{hd}(r_1,r_2,\sigma)$ only if r_1 and r reference the same cell. $\leftarrow_{tl}$ is treated in a similar way.

10,11,12. The definition of equality of references implicitly uses the fact that any reference is equivalent to a "canonical" reference of the form

$$px(\leftarrow(x,r,\sigma)).$$

where $\sim isref(pr)$. The existence of these references is shown in Levy 1978.

The symbol == is used to denote syntactic identity on terms. Thus $t_1 == t_2$ iff t_1 and t_2 are the same expressions.

The problem of errors has not been treated here, but a partial treatment can be found in Levy 1978. For a discussion on the treatment of errors,

see Goguen 1977 and the more practical approaches of Guttag 1977, Tompa 1977. Note that although the type axioms are not in equational form, they can easily be recast into equational form by using the operator if then else on the right sides of the axioms, where this is necessary.

We illustrate the use of the type axioms with some simple examples. We write

$$p_1 \leftarrow r_1;\ \sigma_0$$
$$p_2 \leftarrow r_2;\ \sigma_1$$
$$\vdots$$
$$p_n \leftarrow r_n;\ \sigma_{n-1}$$

to denote the fact that $\sigma_0 = \leftarrow(p_1, r_1, \phi)$ and $\sigma_i = \leftarrow(p_{i+1}, r_{i+1}, \sigma_{i-1})$ for $1 \leq i \leq n$. In addition, it will be assumed that any references appearing in a term r_i are in fact references in σ_{i-1}.

Example

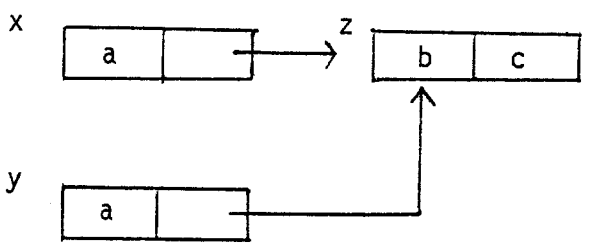
1. Consider the assignments

$x \leftarrow \text{cons}(a,\text{cons}(b,c));$ σ_0

$z \leftarrow t\ell\ x;$ σ_1

$y \leftarrow \text{cons}(a,z);$ σ_2

Pictorially, σ_2 would be represented by



Consider $tl\ y(\sigma_2)$. By axiom 5

$$tl\ y(\sigma_2) = z(\sigma_1)$$
$$= tl\ x(\sigma_0) \qquad \text{by axiom 5 again.}$$

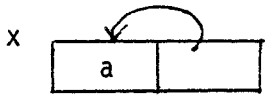
Similarly, $tl\ x(\sigma_2) = tl\ x(\sigma_1)$ by axiom 5

$$= tl\ x(\sigma_0) \qquad \text{by axiom 5.}$$

2. $x \leftarrow cons(a,cons(b,c));$ σ_0

$tl\ x \leftarrow x;$ σ_1

Here σ_1 is represented by



$tl\ tl\ tl\ x(\sigma_1) = tl\ (tl(tl(x(\sigma_1))(\sigma_1))(\sigma_1))(\sigma_1)$

by notational convention.

Now $x(\sigma_1) = x(\sigma_0)$ by axiom 4

and $tl(x(\sigma_0))(\sigma_1) = x(\sigma_0)$ by axiom 9, since $x(\sigma_0) \approx x(\sigma_0)$.

Hence (by symmetry)

$$tl\ tl\ tl\ x(\sigma_1) = x(\sigma_0).$$

3. The above example illustrated the treatment of circularity.

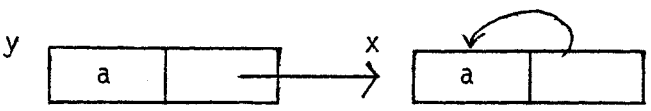
Circularity may also be introduced indirectly.

$x \leftarrow cons(a,b);$ σ_0

$y \leftarrow cons(a,x);$ σ_1

$tltl\ y \leftarrow x;$ σ_2

σ_2 may be represented by



Consider $tl\ tl\ tl\ x(\sigma_2)$.

$x(\sigma_2) = x(\sigma_1) = x(\sigma_0)$ by axioms 4 and 5.

$tl\ x(\sigma_2) = tl\ (x(\sigma_2))(\sigma_2) = tl\ (x(\sigma_0))(\sigma_2)$ by notation and the above argument.

$tl(x(\sigma_0))(\sigma_2) = x(\sigma_1)$ by axiom 9 since

$tl\ y(\sigma_1) = x(\sigma_0)$ by axiom 5

and hence $x(\sigma_0) \approx tl\ y(\sigma_1)$.

Again by symmetry, therefore,

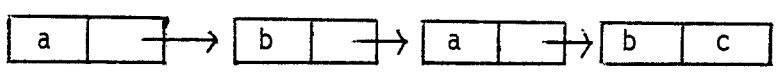
$tl\ tl\ tl\ x(\sigma_2) = x(\sigma_0)$.

4. The same variable may be used more than once.

$x \leftarrow cons(a,cons(b,c))$; σ_0

$x \leftarrow cons(a,cons(b,x))$; σ_1

Diagramatically, σ_1 is



Then $tl\ x(\sigma_1)$ cannot be expressed in terms of σ_0, but

$tl\ tl\ x(\sigma_1) = x(\sigma_0)$ by axiom 5. □

These examples illustrate that references of the form

$$\delta_1(\delta_2(\ldots\delta_n(x(\sigma_{n+1}))(\sigma_n))\ldots)(\sigma_1)$$

can be reduced, by applying the axioms, to references of the form

$qy(\sigma')$

such that σ' is of the form

$$\leftarrow(y,c,\sigma'')$$

and qc is not a reference.

Intuitively, every cell has a canonical reference, namely the shortest "path" to the cell from the variable used when the cell was created.

The axioms for dereferencing are:

14. $\downarrow(\text{cons}(r_1,r_2),\sigma) = \text{cons}(\downarrow(r_1,\sigma), \downarrow(r_2,\sigma))$

15. $\downarrow(a,\sigma) = a$ for all $a \in \text{Atom}$

16. $$\downarrow(px(\leftarrow(x,r,\sigma)),\sigma') = \begin{cases} \downarrow(pr(\sigma),\sigma') & \text{if atom } (pr(\sigma)) \\ \text{cons}(\downarrow(\text{hd } px(\sigma'),\sigma'), (t\ell\ px(\sigma'),\sigma')) & \text{otherwise} \end{cases}$$ □

Example

1. x ← cons(a,cons(b,c)); σ_0

   y ← cons(a,cons(d,e)); σ_1

   tℓ x ← y; σ_2

   hd x ← y; σ_3

$$\downarrow(x(\sigma_3),\sigma_3) = \downarrow(x(\sigma_0),\sigma_3)$$
$$= \text{cons}(\downarrow(\text{hd } x(\sigma_3),\sigma_3), \downarrow(t\ell\ x(\sigma_3),\sigma_3)) \qquad (*)$$

Now

$$\downarrow(\text{hd } x(\sigma_3),\sigma_3) = \downarrow(y(\sigma_1),\sigma_3) \quad \text{by reducing hd } x(\sigma_3)$$
$$= \text{cons}(\text{hd } y(\sigma_3), t\ell\ y(\sigma_3))$$
$$= \text{cons}(a,\text{cons}(d,e))$$

and similarly we can show that

$$\downarrow(t\ell\ x(\sigma_3),\sigma_3) = \text{cons}(a,\text{cons}(d,e))$$

Hence by (*) $= \text{cons}(\text{cons}(a,\text{cons}(d,e)),\text{cons}(a,\text{cons}(d,e)))$.

2. $x \leftarrow \mathrm{cons}(a,b);$ σ_0

$t\ell\ x \leftarrow x;$ σ_1

$$\begin{aligned}
\downarrow(x(\sigma_1),\sigma_1) &= \downarrow(x(\sigma_0),\sigma_1) \\
&= \mathrm{cons}(\downarrow(\mathrm{hd}\ x(\sigma_1),\sigma_1),\ \downarrow(t\ell\ x(\sigma_1),\sigma_1)) \\
&= \mathrm{cons}(\downarrow(\mathrm{hd}\ x(\sigma_0),\sigma_1),\ \downarrow(x(\sigma_0),\sigma_1)) \\
&= \mathrm{cons}(a,\downarrow(x(\sigma_0),\sigma_1)) \\
&= \mathrm{cons}(a,\mathrm{cons}(a,\downarrow(x(\sigma_0),\sigma_1)) \quad \text{etc.}
\end{aligned}$$

□

4. VERIFICATION

The proof rule for assignment, discussed in section 1 is $Q^{\sigma}_{U(\sigma,p,\ell)}\{p \leftarrow \ell\}\ Q$. We illustrate this rule with the text and the assertions of a simple program taken from Burstall 1972. Following the notational suggestion of section 1, we write σ' to denote $U(\sigma,p,\ell)$ provided that p and ℓ can be determined by context. Note that $U(\sigma,p,\ell)$ can now be defined formally, namely $U(\sigma,p,\ell) \triangleq \leftarrow (p,\ell,\sigma)$.

The following simple program reverses a non-circular list σ_0, referenced by k. $\{Q_0\}$ thru $\{Q_7\}$ are assertions given below.

reverse(k,j); $\{Q_0\}$
j ← nil; $\{Q_1\}$
<u>while</u> k ≠ nil <u>do</u> $\{Q_2\}$
    i ← tℓ k; $\{Q_3\}$
    tℓ k ← j; $\{Q_4\}$
    j ← k; $\{Q_5\}$
    k ← i; $\{Q_6\}$
<u>end</u> $\{Q_7\}$

Let $TL = \{t\ell\}$. Then TL^* is the free monoid on TL, that is expressions of arbitrary finite length composed of $t\ell$. Let λ denote the empty string, and define $TL^+ = TL^* - \{\lambda\}$. If $p \in TL^*$, then the length of p, $|p|$ is defined by $|\lambda| = 0$ $|t\ell\ p| = 1 + |p|$. We write $t\ell^n$ to denote the element $p \in TL^*$ such that $|p| = n$. Hence $t\ell^0 = \lambda$. The interpretation of a term $r_1 = r_2$ where $r_1, r_2 \in Ref$ is simply congruence of r_1 and r_2 with respect to the type axioms.

Now the requirement for the program reverse is that the list referenced

by k be reversed "in place" and the resultant list be referenced by j. It is possible to state this assertion in terms of the elements in the head of each cell of the list, as is done for example in Burstall, but this in itself does not express the fact that the reversal is in place. Instead, we define for σ,

$$\text{Reverse}(k,j) \equiv \exists m \forall n < m \cdot t\ell^{m-n-1}j(\sigma) = t\ell^{n}k(\sigma_0)$$

to denote the fact that the list referenced by j in σ is the in place reverse of the list referenced by k in σ_0. ($n,m \in N^+$ the set of non-negative integers). This is not quite sufficient for the post-condition, however, since j might not be at the "end" of the list. Hence we need also $t\ell^{m}k(\sigma_0) = \text{nil}$. Thus we have

$$Q_7: \exists m \forall n < m \cdot t\ell^{m-n-1}j(\sigma) = t\ell^{n}k(\sigma_0) \;\&\; t\ell^{m}k(\sigma_0) = \text{nil}.$$

The following predicates are also needed for the proof:

Unchanged(k): $\forall p \in TL^* \cdot p(k(\sigma))(\sigma_0) = pk(\sigma)$

Distinct(j,k): $\forall p_1,p_2 \in TL^* \cdot p_1 j(\sigma) \neq p_2 k(\sigma)$

Noncircular(k): $\forall p \in TL^+ \cdot pk(\sigma) \neq k(\sigma)$.

Unchanged(k) expresses the fact roughly that the tails of each cell in the list referred to by k in σ are not changed from their original values. Distinct(j,k) says that there is not sharing between j and k. Finally, Noncircular(k) says that the list k is not circular. The assertions are

Q_0: Noncircular(k)

Q_1: Unchanged(k) & Noncircular(k) & $j(\sigma) = \text{nil}$

Q_2: (The invariant) Unchanged(k) & Noncircular(k) & Distinct(j,k) & Reverse(k,j)
& $j(\sigma) \neq \text{nil} \Rightarrow t\ell\,(j(\sigma))(\sigma_0) = k(\sigma)$

Q_3: Q_2 & $i(\sigma) = t\ell\ k(\sigma)$

Q_4: Unchanged(i) & Noncircular(i) & Distinct(k,i) & Reverse(k,k)

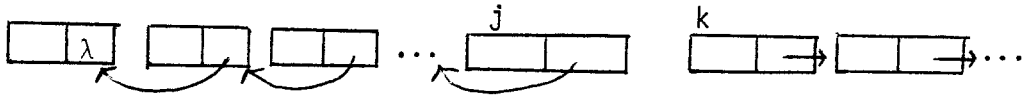
& $t\ell\ (k(\sigma))(\sigma_0) = i(\sigma)$

Q_5: Q_4 & $j(\sigma) = k(\sigma)$

Q_6: Q_2 (the invariant)

Q_7: as given above

Pictorally, the invariant Q_2 is



The expression $t\ell(j(\sigma))(\sigma_0) = k(\sigma)$ illustrates the flavour of the assertion: it asserts that the tail of the cell referred to in σ by j originally contained a reference to the cell now referred to by k. The proof of this program is straightforward, and (usually) proceeds by showing that $Q_i \Rightarrow Q_{i+1}(\sigma')$ where $Q(\sigma')$ denotes $Q^{\sigma}_{\sigma'}$. Consider the verification of the program segment

$$\{Q_2\}\ \ i \leftarrow t\ell\ k\ \{Q_3\}$$

We must show that

$$Q_2 \Rightarrow Q_3(\sigma')$$

where $Q(\sigma') = Q^{\sigma}_{\sigma'}$, and $\sigma' = \leftarrow(i, t\ell\ k, \sigma)$. Now $Q_3(\sigma') = Q_2(\sigma')$ & $i(\sigma') = t\ell\ k(\sigma')$ by definition. Since i does not appear in Q_2, we can clearly apply axiom 5 to each term in $Q_2(\sigma')$, and hence $Q_2(\sigma') = Q_2$. Furthermore

$i(\sigma') = t\ell\ k(\sigma)$ by axiom 5

and $t\ell\ k(\sigma') = t\ell\ k(\sigma)$ also by axiom 5.

Hence $Q_2 = Q_2(\sigma')$

$\Rightarrow Q_2(\sigma') \;\&\; t\ell\ k(\sigma) = t\ell\ k(\sigma)$

$\Rightarrow Q_2(\sigma') \;\&\; i(\sigma') = t\ell\ k(\sigma')$

$= Q_3(\sigma')$ as required.

The program segment

$$\{Q_3\}\ t\ell\ k \leftarrow j\ \{Q_4\}$$

is more interesting: axioms 4 and 9 must be used. We must show that

$$Q_3 \Rightarrow Q_4(\sigma').$$

Consider for example the portion of $Q_4(\sigma')$ obtained from Reverse(k,k). That is, we must show that

$$Q_3 \Rightarrow \exists m \forall n < m \cdot t\ell^{m-n-1}k(\sigma') = t\ell^n k(\sigma_0).$$

To do this, we need the following clause from Q_3:

$$\text{Reverse}(k,j) \;\&\; \text{Distinct}(j,k) \;\&\; t\ell\ (j(\sigma))(\sigma_0) = k(\sigma).$$

Note that $\text{Distinct}(j,k) \Rightarrow pj(\sigma) \neq qk(\sigma')$ for any $p,q \in TL^*$. Now

$t\ell^{m-n-1}(t\ell\ k(\sigma'))(\sigma') = t\ell^{m-n-1}(j(\sigma))(\sigma')$ by axiom 9

$= t\ell^{m-n-1}(j(\sigma))(\sigma)$ by axiom 9

since $\text{Distinct}(j,k)$.

Hence $Q_3 \Rightarrow \exists m \forall n < m \cdot t\ell^{m-n-1}j(\sigma) = t\ell^n k(\sigma_0)$ by definition of Reverse(k,j) (*)

$\Rightarrow \exists m \forall n < m \cdot t\ell^{m-n}k(\sigma') = t\ell^n k(\sigma_0)$ by the above derivation.

Thus $\exists m' \forall n < m'-1 \cdot t\ell^{m'-n-1}k(\sigma') = t\ell^{n}k(\sigma_0)$

where $m' = m+1$.

But when $n = m'-1$, we have

$$t\ell^{m'-n-1}k(\sigma') = t\ell^{n}k(\sigma_0)$$

since $t\ell^{m'-n-1}k(\sigma') = t\ell^{0}k(\sigma') = k(\sigma') = k(\sigma)$ by axiom 4

$= t\ell(j(\sigma))(\sigma_0)$ from Q_3

$= t\ell^{m}k(\sigma_0)$ by an instantiation of

$= t\ell^{m'-1}k(\sigma_0)$ as required.

Hence $Q_3 \Rightarrow \exists m' \forall n < m' \cdot t\ell^{m'-n-1}k(\sigma') = t\ell^{n}k(\sigma_0)$.

Similar reasoning can be used to verify the remaining assertions and hence obtain $Q_3 \Rightarrow Q_4(\sigma')$ as required. With experience, and when confidence with the axioms has been gained, the assertions can be verified in less detail. Of course the difficulty of finding the invariants of the program remains, however the power of the language allows for much more ease in expressing structures - consider for example, the ease of relating values of a structure σ to an "initial" structure σ_0. The method is also not restricted to any particular class of programs, but may be applied to any program using the type.

5. CONCLUSIONS

We have presented an abstract model of a type linked list which allows sharing and a completely general assignment. We have presented axioms for the type and given an indication of the properties of the model in terms of algebras and quotients. It is possible to use this model to express naturally and succinctly such concepts as "in place" changes to a data structure and to prove properties of programs manipulating the type. The model differs from previous models in the way referencing is treated abstractly and, because of the treatment, it becomes possible to reason about sharing without having to resort to a notational device which makes the sharing explicit. We believe that the semantic model will thus lead to simpler proof methods for programs with referencing.

ACKNOWLEDGEMENTS

The work by Goguen, Thatcher, Wagner and Wright has a strong influence on the idea of abstractions expressed in this paper. The author would also like to thank Professors E.A. Ashcroft, T.S.E. Maibaum, R.M. Burstall and R. Milner for their valuable suggestions.

REFERENCES

1. ADJ 1977a: Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.G., Initial Algebra Semantics and Continuous Algebras, JACM, Vol. 24, No. 1 (Jan. 1977).

2. ADJ 1977b: Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.G., An Initial Algebra Approach to the Specification Correctness and Implementation of Abstract Data Types, IBM Research Report RC 6487.

3. Barron, D.W., An introduction to the study of programming languages, Cambridge, C.Sc. Texts, 1977.

4. Burstall, R.M., Some Techniques for Proving Correctness of Programs which Alter Data Structures, Machine Intelligence 7, Edinburgh, 1972.

5. Guttag, J., Abstract Data Types and the Development of Data Structures, CACM, Vol. 20, No. 6 (June 1977).

6. Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No. 10 (Oct. 1969).

7. Kowaltowski, T., Data Structures and Correctness of Programs, Technical Report Departmento de Mathematica Aplicada, Universidade de Sao Paulo, 1976.

8. Levy, M.R., Some Remarks on Abstract Data Types, Sigplan Notices, Vol. 12, No. 7 (July 1977).

9. Levy, M.R., Data Types with Sharing and Circularity, Ph.D. Thesis, University of Waterloo, Department of Computer Science, Technical Report CS-78-26 (May 1978).

10. McCarthy, J. et al, The Lisp 1.5 Programmers Manual, MIT Press, Cambridge, (1962).

11. Oppen, D.C. and Cook, S.A., Proving Assertions about Programs that Manipulate Data Structures, Proc. 7th Annual ACM Symposium on Theory of Computing, New Mexico, 1975.

12. Reynolds, J.C., Reasoning About Arrays, (To Appear).

13. Scott, D. and Strachey, C., Towards a Mathematical Semantics for Computer Languages, Proc. Symp. on Computers and Automata, Microwave Research Institute Symposium Series, Vol. 21, Polytechnic Institute of Brooklyn, 1972.

14. Tompa, F.W., A Practical Example of the Specification of Abstract Data Types, Tech. Report, Departamento de Informatica, Pontificia Universidade Catolica do Rio de Janeiro (1978).