DATA TYPES WITH SHARING AND CIRCULARITY

Michael R. Levy

Department of Computer Science
University of Waterloo
Waterloo, Ontario

CS-78-26
May 1978

DATA TYPES WITH SHARING AND CIRCULARITY

by

Michael Robert Levy

A thesis
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1978

DEDICATION

To my parents Laurie and Ruth Levy.

DATA TYPES WITH SHARING AND CIRCULARITY

by

MICHAEL ROBERT LEVY

A B S T R A C T

Initial algebraic techniques such as the algebraic specifications of data types have proved useful in characterizing data types. In this thesis, we treat concepts of sharing and circularity in data using an algebraic approach.

Firstly, it is shown how the idea of congruence can be used to give the specification for a data type which has referencing. The semantics of variable naming and a general assignment is also given by the specification. A proof rule for this type is illustrated by verifying a simple program. It is also shown how assignment and variables can be treated for other types which are specified algebraically.

Secondly, the algebraic specification technique is extended to continuous algebras, and it is shown how types with infinite objects can be characterized as continuous types. Lists, including infinite lists, are treated in this way. It is shown how sharing and circularity can be placed in this framework, by characterizing data structures as a set of regular equations, where the equations capture the sharing in the structure. The value of the structure is viewed as the solution of the equations. Data types with sharing and circularity are then defined in terms of these structures, and it is shown how this can be done for a type whose objects are abitrary list structures with the sharing and circularity. This type is continuous, and hence the usual techniques available for the treatment of continuous spaces may be used. The proof of a simple recursive program which manipulates a list structure is given.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Firstly, I would like to express my appreciation to my supervisors, Professor T.S.E. Maibaum and Professor E.A. Ashcroft, for their encouragement, suggestions and advice. I would also like to thank Professor F.W. Tompa for his valuable suggestions and comments. Professor Ashcroft suggested the idea of representing shared structures by sets of equations and of defining updating in terms of the equations.

I would also like to thank my wife, Diane, for her support and patience during the course of my studies.

The manuscript was typed by Olga Reibling, assisted by Kathy Knight and Heather Pente.

Finally, I wish to acknowledge the financial support of the University of the Witwatersrand Council, the University of Waterloo and the National Research Council of Canada.

Chapter 1

Introduction

Much activity has revolved recently around the idea of data types. The notion of types pre-dates computers (Church [10], 1940), but computing forces a particular perspective on anyone contemplating ways of organizing or "typing" data. Several novel ways of organizing data were seen to offer substantial economies in performing operations in this data (See Knuth [24]) and the search for new ways to represent data is still being conducted with vigour. It has also proven to be important to study the more conceptual aspects of data types. Firstly, from a practical standpoint, the natural question of how to implement newly conceived data types arises. It is also interesting to ask if the design of a data type can aid in the development of a programming project. The concept of data types provides a natural way to extend the idea of modularity from routines to data.

From a theoretical point of view, a number of interesting questions arise. The most fundamental of these is the question "What is a data type?" The answer to this question can have an important impact on the ease of answering other questions about the type. It has long been recognized that a data type is in some sense independent of a particular representation. (Earley [13], Hoare [20], Standish [39]). Thus natural questions to ask are: "How can a type be specified in a way that is independent of any representation?" and "How can an implementation be shown to be correct?" The first question is of much more than theoretical interest, for it is at the heart of the concept of modularity. The issue is very similar to that of application-independence in data base systems, where the particular way that data is represented is considered to be "invisible" to application programs. The second question (namely of correctness of implementation) is important since correctness is essential for all other aspects of an implementation to be meaningful.

The attempt to answer the questions posed above has proved to be difficult. It has become clear that a data type is more than a collection of structured objects: in fact the operators of a type are, in many cases, of more interest than the objects themselves. The reason for this is that any attempt to "examine" the data objects may force us to choose some representation - something we are trying to avoid. By considering only the "behavioural" aspects of a type, that is, the effect of operations on objects, it is possible to achieve a strong degree of representation independence. Data types can thus be viewed profitably as a set of objects and a set of operations on these objects, in other words, as an algebra.

There are several practical questions which any useful theoretical approach should attempt to answer: How is a data type characterized? How is it specified? Is the specification correct? Is it possible to prove the correctness of an implementation of the type? How can programs using the type be verified? Some of these problems were tackled by Earley [13] and Standish [39]. More recently, investigation of the "algebraic" approach has been done by ADJ [1] and Guttag [18]. On a more practical level, several new languages have been proposed which allow for a convenient way to implement types as algebras. These include CLU (Liskov et al [28]), Alphard (Wulf et al [41]), Euclid (Popek et al [34]) and Gypsy (Ambler et at [4]). The idea of representation independence has come to be known as "abstractness": a data type is abstract if it is defined independently of any actual implementation. In the algebraic approach the concept of isomorphism is used to capture this notion of abstractness. A data type is defined by describing an abstract data type (algebra), and an implementation is correct if, when it is treated as an algebra, it can be shown to be isomorphic to the abstract data type. Similarly, the abstract data type itself can be verified by showing that it is isomorphic to an algebra which we already believe to be an implementation of the type. The word "implementation" has been used very loosely here. We may regard any concrete description of an algebra as being an implementation of some data type. In terms of actual machine implementations or implementations of one data type in terms of other types, there are difficulties with this rather simple view of isomorphism. For a further discussion of these problems, see ADJ [1] and Lehman and Smyth [26].

There is some consensus that the algebraic view of data types is reasonable, at least for a large class of data types. Some problems do arise, however. These include the problems of dealing with error conditions and the problem of verifying that implementations are correct. Neither of these topics are discussed in this thesis, but are covered in Goguen [16], ADJ [1], Guttag et al [17] and Lehman and Smyth [26]. When data types are used in programming languages, they are usually used with variables and assignment to the variables, but the semantics of this assignment is often omitted from the formal specification of the data type. Thus the implementor may decide, for example, whether to implement the type by allowing the functions to have side effects or not, and must also in fact decide on the exact format of assignment to variables. This is counter to the idea of abstract data types since it makes the programs dependent on the particular mechanism chosen for assignment. Guttag et al [17] discuss this problem and suggest a practical solution for overcoming it. But when considering data types with sharing, the proper formal treatment of assignment becomes critical.

We show in this thesis how variables and assignment can be treated as part of the type, and hence be included in the specification. The problem of data types with sharing has received very little attention, either in the context of abstract data types or in a more general setting. A notable exception to this is the pioneering paper by Burstall [9] describing a method of proving correctness of programs which alter data structures. This paper concentrated more on a proof system then on the semantics of the type, whereas this thesis is mainly concerned with the semantics of sharing. The first part of the thesis (chapter 3) shows how sharing (and a special type of sharing called circularity) can be accommodated in the setting of abstract data types *a là* ADJ[1]. The second part (chapter 4) discusses sharing and circularity in the setting of continuous algebras rather than all algebras.

Sharing of data raises several problems, both in practise and in theory. In an implementation of a type, two structures may share part of the storage used in the representation of the structure. If this is the case, then updates to one object may affect another object. This creates difficulties, especially in terms of verification of programs using types with shared variables. Is it possible to formulate a proof rule when program variables may be affected in an apparently undisciplined way? Examples of sharing arise most naturally in the type list, such as the underlying type of LISP (McCarthy et al [32]). Only a very primitive form of sharing was allowed in Pure LISP, but general sharing was permitted in LISP 1.5 by use of the operators REPLACA and REPLACD. Although these operators are easily implemented, it was apparently not easy to include them in the LISP formalism, and they were never defined formally. Several authors have recognized the difficulties associated with sharing and some have proposed that sharing be disallowed (Hoare [21], Kieburtz [23]). A compromise proposal for treating references has been suggested by the designers of Euclid [34], where references must be considered part of the type being referenced. This proposal fits in easily with the abstract model of linked lists developed in this thesis. We do not propose here the undisciplined use of sharing, but do suggest that sharing is a useful concept, and the problems associated with it cannot be solved by ignoring them.

When considering sharing we must often also consider circularity - for example, we may imagine that an object of some type shares a component with itself. Once again, lists provide the most natural example of such sharing, as in for example circular lists.

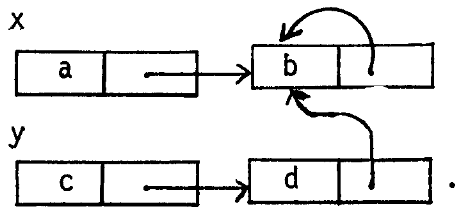
What we attempt to show in this thesis is that the algebraic approach to data types does lend itself to an elegant treatment of types with sharing. In chapter 3 of this thesis, the type linked list is considered. This type has as objects arbitrary list structures (binary graphs, although arbitrary graphs could be considered), and its operators include the usual list operators as well as assignment to cells in the structures. (This generalized assignment corresponds to REPLACA and REPLACD of LISP, but we have chosen the more representation independent notation of Burstall [9]). This type has the property that it is easy to implement, and yet seemingly difficult to formalize. There are "operational" formulations of the type, such as the Scott-Strachey approach [38] with functions which, given a "location", return a "value". The value itself may be complex, and include other locations. This is the usual definitional model used for linked lists, but is not very different from operational descriptions of the type. In order to verify programs using the type with this semantic model, it is necessary to use complex rules or notational devices, or else to restrict attention to a subclass of all possible list structures. (Burstall follows the latter route [8]. Kowaltowski [23] considers arbitrary list structures, but must introduce fairly complex notational devices for proofs. Oppen and Cook [31] base their semantics on a graph theoretic model, which is also operational, and formulate extremely complex proof rules for the type.) With a more "abstract" treatment of the notion of "pointing" or referencing, it is possible to develop the right insight in order to tackle the difficulties presented by sharing.

Although we have considered in detail only the type linked list, the general nature of the type allows a simple extension of the techniques used for any other type with referencing. We illustrate in chapter 5 that an extremely simple proof rule (a backward-substitution rule) can be used for the purposes of program verification. The abstract specification of the type suggests a general formalism for direct reasoning about the program. We believe that this formalism is as expressive as any more operational language and because of its abstract nature it is more natural to use for verification. The idea of using congruences as a means of characterizing types abstractly has, in previous works on algebraic specification techniques, been used, more in order to tackle the questions of correctness of specifications and implementations, than in actually considering the nature of the types themselves. We show that the same considerations about "abstractness" that led to the initial quotient characterization of data types leads us to an elegant way to characterize the notion of reference and hence to define the type linked list. Whereas initiality was perhaps the crucial idea for general characterizations, it is the idea of congruence and quotient which turns out to be essential for an abstract treatment of linked lists.

A further advantage of the specification of linked list given in chapter 3 is that we have used a standard technique of specification (namely algebraic specification), rather than inventing a new descriptional method. Similarly it is straightforward to use the proof rule for verification with any of the common verification techniques, such as invariant assertions or intermittent assertions, (Floyd [15], Hoare [19],Burstall [8],Manna and Waldinger [31]).

In chapter 4, sharing is considered in a somewhat more general sense than chapter 3. In that chapter the setting is continuous algebras rather than arbitrary algebras. It has been shown for example how circular lists may be considered as being in fact infinite elements of a continuous domain (Reynolds [35]). However, several difficulties present themselves with this approach. Firstly, when generalized assignment is added to the type, it becomes important to know more about the structure of the list than is afforded by its infinite "descriptions". If types with sharing are to be allowed, and attention is not just restricted to non-shared structures, then this problem becomes critical. A second problem is to try to find a characterization for data types that are continuous which can be used to answer the same questions as those posed for the non-continuous types discussed above. Unfortunately, the results do not all follow through directly, but they do hold in certain special cases. We snow in chapter 4 firstly a condition which will allow the characterization of continuous types as quotients of the initial algebra in the appropriate class of continuous algebras. Next, we present the axioms of a general type called List, whose objects may be considered to be finite or infinite lists of the Pure Lisp sort, and then show that the type is in fact characterized in the usual way by a quotient algebra which is initial in the class of all continuous algebras satisfying the axioms. We believe that this treatment is more conceptually pleasing than the "inverse limit" construction for the same objects presented in Reynolds [35].

We view the type axioms as including some structure on the initial continuous algebra by taking a quotient of this algebra. Thus two expressions over the operators of the type denote the same element of the type if and only if the terms are in the same class in the congruence used to take the quotient. This then is the notion of abstractness - an object of the type can be considered abstractly as the collection of all well formed expressions which in some sense yield that object. The utility of this view is not debated here, since it may be found elsewhere. (See ADJ [1]). The next step is to introduce shared lists and show how they may be treated. Firstly we note that the sharing in some particular structure may be represented by a set of regular equations. For example, suppose we have two lists named x and y which share their last elements. Suppose also that this last element is circular. Pictorially, we may have



To represent this structure with equations, we introduce a new variable z (which names the shared sublist) and write

$$x = \mathrm{cons}(a,z)$$
$$y = \mathrm{cons}(c,\mathrm{cons}(d,z))$$
$$z = \mathrm{cons}(b,z).$$

Such a set of equations is called a structure. The intention is that there is a close correspondence between the set of equations and the structure we have in mind. For example, the number of times the operator "cons" appears on the right hand sides of the set of equations must be the same as the number of boxes appearing in a diagram of the type. Other properties of these equations are discussed in chapter 4.

Note that if the type axioms for Lists can be viewed as inducing some structure on the set of expressions of the type, then a particular structure represented as a set of equations (regular) can be viewed as inducing additional structure on the expressions. For example, in the above set of equations we would expect that $t\ell\ x = t\ell\ t\ell\ y$. This effect is achieved technically by considering the set of equations as if they were axioms defining the "constant" or "nullary" symbols called program variables. We show that the conditions needed for the existence of an initial quotient will always exist when regular equations are added to the type. In addition, we show how the solution to the structure equations can be used to characterize the equivalence induced by the structure. The final step is to define an algebra that allows for the manipulation of these sets of equations. We define such an algebra, and show that it is continuous. In chapter 5 we show how a standard induction technique, the stepwise computational induction of Scott and De Bakker, described in Manna [30], can be used for purposes of verification of a recursive function which manipulates a list. We use for the proof a language based on the concept of sets of equations characterizing sharing.

The thesis is outlined below. In chapter 2, the mathematical preliminaries are presented. We have used throughout the thesis the notation of ADJ, and most of the results in chapter 2 are from ADJ [1] or [2]. We also propose a new definition of congruences, called continuous congruences, and show that there is always a least continuous congruence of a continuous algebra containing an arbitrary binary relation. Further properties of these congruences have not been investigated in this thesis. In chapter 3 we outline some of the difficulties of abstract data types specification techniques, and suggest how these difficulties may be overcome. We then present, with motivation, the type linked list. We show how programs using this type may be verified by presenting a proof rule for the type. In practice, properties of types are often best proven using a canonical term algebra which is a representative algebra of the type.

We define such an algebra for linked list, and show that it is indeed canonical. Finally, we present an operational model of the type, and show that it satisfies the axioms. In chapter 4 the program of generalizing the results of finite types to continuous types is carried out. We also present the continuous type LIST as outlined above. Finally in chapter five we illustrate the techniques of verification of programs suggested by the models in chapters 3 and 4, and then make suggestions for future study.

Chapter 2

Mathematical Preliminaries

1 Many-sorted algebras and initiality

A data type is viewed in this thesis as a many-sorted algebra. (For a general discussion of algebras see Cohn [11]). Discussion of data types as many-sorted algebras can be found in ADJ [1], Guttag [18] and Levy [27]. 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 called the members of the carrier. 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, and we define a data structure to be an element of the carrier of a data type. The notation and results are from ADJ [1,2].

Definition 1 Let S be a set whose elements are called sorts. An S-sorted operator domain Σ is a family of sets $\Sigma_{w,s}$ of symbols, for $s \in S$ and $w \in S^*$ where S^* is the free monoid on S. $\Sigma_{w,s}$ is the set of operator symbols of type $<w,s>$, arity w and sort s.

A Σ-algebra A consists of a family $\langle A_s \rangle_{s \in S}$ of sets called the carrier of A, and for each $\langle w,s \rangle \in S^* \times S$ and each $\sigma \in \Sigma_{w,s}$ a function

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

(where $w = s_1 s_2 \ldots s_n$) called the operation of A named by σ. (If $w = s_1 s_2 \ldots s_n$, then let A^w denote $A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n}$.) $\square$

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 $s \in S$, we usually denote the set A_s by s.

Definition 2 If A and A' are both Σ-algebras, then a Σ-homomorphism h: A $\to$ A' is a family of functions

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

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

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

Definition 3 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. $\square$

<u>Theorem 1</u> The class of all Σ-algebras has an initial algebra called T_Σ. □

(T_Σ is sometimes called the <u>free</u> Σ-algebra.) This theorem, as well as the others given in this section, is proven in ADJ [1]. T_Σ can be viewed intuitively as the algebra of well-formed expressions over Σ.

<u>Variables</u> can be included in terms in T_Σ in the following way.

<u>Definition 4</u> 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 <u>variables of sort</u> s. Suppose Σ is an S-sorted operator domain. Let

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

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

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

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

Thus variables are 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.

Definition 5 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. $\square$

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

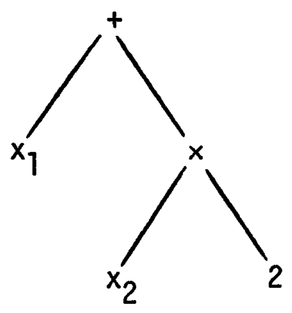
$$\bar{\theta}: \ T_\Sigma(X) \to A$$

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

The intuition for the above objects is that the initial algebra in a class of algebras is the algebra consisting of "syntactic" terms over the operators. It can be viewed as the set of trees denoting possible well-formed expressions in the appropriate class of algebras. Note that this analogy is perhaps too strong in the sense that the trees mentioned above are abstract trees: their concrete representation could be as diagrams, polish, infix or any other notation.

$T_{\Sigma}(X)$ has as carrier trees similar to T_{Σ}, but the leaves can be constants or variables (elements of X). Theorem 2 says that if we give variables some value in a Σ-algebra, then there is a unique way to evaluate any terms from $T_{\Sigma}(X)$ in A.

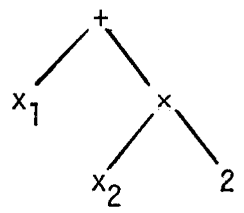
For example, let $S = \{\underline{\text{integer}}\}$, $\Sigma_{\lambda,\underline{\text{integer}}} = \{0,1,\ldots\}$, $\Sigma_{\underline{\text{integer}}\ \underline{\text{integer}},\ \underline{\text{integer}}} = \{+,\times\}$ and $X = \{x_1,x_2\}$. Also, let N denote the usual algebra of non-negative integers. The initial Σ-algebra is the algebra consisting of "abstractly well-formed" expressions. Now consider the following tree in $T_{\Sigma}(X)$:



Theorem 2 tells us that given any assignment to x_1,x_2 in N (for example $x_1 = 3$, $x_2 = 6$), there is only one possible value in N corresponding to the tree (in the example it is 15).

<u>Definition 6</u> If $< X_s >_{s \in S}$ is a family of variables, then each $x \in X_s$ has arity λ and sort s. Let $X_n = \{x_1, \ldots, x_n\}$ be a set of n variables. Then $t \in T_\Sigma(X_n)$ is called a <u>Σ-term in n variables</u> or an n-ary Σ-term. $\square$

The motivation for this definition is that any term $t \in T_\Sigma(X_n)$ can be viewed as a function of n arguments whose value is the value of the tree after substituting each argument for its corresponding variable in the tree. So for example the tree



defines a function $t_A\colon\ A^2 \to A$ for any Σ-algebra A.

<u>Definition 7</u> Given a Σ-algebra A and $t \in T_\Sigma(X_n)$ we define a corresponding <u>derived operator</u> on A

$$t_A\colon\ A^w \to A_s \qquad \text{where } w = s_1, s_2, \ldots, s_n$$

and sort(t) = s, as follows:

Suppose $<a_{s_1}, a_{s_2}, \ldots, s_{s_n}> \in A^w$, and also that $\text{sort}(x_i) = s_i, 1 \le i \le n$.

Let $a\colon\ X_n \to A$ be defined by

$$x_i \mapsto a_{s_i} \qquad 1 \le i \le n$$

where $a_{s_i} \in A_{s_i}$, $1 \le i \le n$. Then by theorem 2 there exists a unique Σ-homomorphism.

$$\bar{a}:\ T_\Sigma(X_n) \to A$$

extending $a:\ X_n \to A$.

So given $t \in T_\Sigma(X_n)$ define

$$t_A(a_{s_1},\ldots,a_{s_n}) = \bar{a}(t) \qquad \square$$

<u>Definition 8</u> A <u>Σ-equation</u> 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 <u>satisfies</u> e if

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

for <u>all</u> assignments $\theta:\ X \to A$. If ε is a set of Σ-equations, then A satisfies ε iff A satisfies each $e \in \varepsilon$. $\square$

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 $\underline{\underline{\mathrm{Alg}}}_{\Sigma,\varepsilon}$.

<u>Definition 9</u> An <u>equational specification</u> 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 <u>type axioms</u>).

The class of algebras $\underline{\underline{\mathrm{Alg}}}_{\Sigma,\varepsilon}$ does in fact have an initial algebra, denoted $T_{\Sigma,\varepsilon}$ and it is this algebra (or one isomorphic to it) which is usually regarded as the data type defined by a set of equations. The concept of "abstract" is captured algebraically by isomorphism, since different algebras (that is, algebras with different representations) may be isomorphic. In fact the structure of $T_{\Sigma,\varepsilon}$ can be characterized as an algebraic quotient of T_Σ where intuitively two elements of T_Σ are equivalent if and only if one can be derived from the other by using the equations. That is $T_{\Sigma,\varepsilon}$ groups together all equivalent terms.

An important concept in the theory of abstract data types is the idea of quotients mentioned above. A quotient partitions the carrier of an algebra, and when this quotient is over T_Σ, it can be interpreted as a way of equating syntactic terms over the alphabet of the type. The importance of such "equations" is that they provide a means for expressing the difficult concept of abstraction. Furthermore quotients are defined in terms of equivalence relations which are congruences; intuitively, terms that have been equated must behave in the same way with respect to the operators of the type.

<u>Definition 10</u> A <u>Σ-congruence</u> $\equiv$ on a Σ-algebra A is a family $< \equiv_s >_{s \in S}$ of equivalence relations $\equiv_s$ on A_s such that if $\sigma \in \Sigma_{w,s}$ where $w = s_1 s_2 \ldots s_n$ and $a_i, a_i' \in A_{s_i}$ and if $a_i \equiv_{s_i} a_i'$ for $1 \leq i \leq n$, 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 $(A/\equiv)_s = A_s/\equiv_s$ be the set of $\equiv_s$-equivalence classes of A_s for each $s \in S$. For $a \in A_s$ let $[a]_s$ denote the $\equiv_s$-class containing a. It is possible to make $A/\equiv \; = \langle A_s/\equiv_s\rangle_{s \in S}$ 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 $[a_i] \in A_{s_i}/\equiv_{s_i}$ for $1 \leq i \leq n$

then

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

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 $\varepsilon = \{\langle L,R\rangle | L,R \in T_\Sigma(X)\}$ generates a binary relation $R \subseteq A \times A$. This relation is the set of all pairs $\{\langle\bar{\theta}(L), \bar{\theta}(R)\rangle \mid \theta$ is an assignment$\}$.

Theorem 3 If A is a Σ-algebra and R is a relation on A, then there exists a least Σ-congruence relation on A containing R; it is called the congruence relation generated by R on A. (The ordering on Σ-congruences is the subset ordering.) □

Theorem 4 If ε is a set of Σ-equations generating a congruence q on T_Σ, then T_Σ/q is initial in $\underline{\underline{\mathrm{Alg}}}_{\Sigma,\varepsilon}$. □

The importance of the above theorems is that any set of Σ-equations (axioms) "automatically" defines an algebra which can be regarded as the symbolic model of the object being defined. This model can be used to answer such questions as "Do the axioms characterize some particular model of the type?" and "Is a given implementation of the type correct?" In chapters 3 we use the model to gain insight into the abstract notion of references in structures with sharing. Specifications can be "proven correct" by showing that for some Σ-algebra M which is a model of the type, $M \cong T_\Sigma/q$. In practice we may use a canonical algebra C where $C \cong T_\Sigma/q$ and show that $M \cong C$. C is a "term" algebra-that is, each $c \in C$ is in T_Σ.

Definition 11 A Σ-algebra C is a canonical Σ-term algebra if

(i) $C_s \subseteq T_{\Sigma,s}$ for each $s \in S$

and (ii) if $\sigma(t_1 \ldots t_n) \in C$ then $t_i \in C$ for each i, $1 \leq i \leq n$, and $\sigma_C(t_1 \ldots t_n) = \sigma(t_1 \ldots t_n)$. □

Theorem 5 Let $<s, \Sigma, \varepsilon>$ be a specification and let C be a canonical Σ-term algebra. Then

$$C \cong T_{\Sigma,\varepsilon}$$

iff

(i) C satisfies ε

and (ii) for each $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, $t_i \in C_{s_i}$ $1 \leq i \leq n$,

$$(\sigma(t_1 \ldots t_n), \sigma_C(t_1 \ldots t_n)) \in q$$

where q is the congruence relation on T_Σ generated by ε. □

2 Continuous Algebras

There is a class of data types such as circular lists which may be naturally regarded as being characterized by "infinite" expressions. The definitions above about T_Σ can be extended to an algebra CT_Σ whose objects can be thought of as finite or infinite expressions over Σ. In order to carry out this extension we consider partially ordered sets, with the interest here in data types. The idea of using partially ordered sets as a data domain is in fact used in Scott's work on semantics [37], but there the concern is more with procedures as elements of a data space than with the more usual data types. However, Reynolds [35] has shown how complex data types such as lists might be considered as partially ordered sets (in fact, as complete lattices). Intuitively speaking, the order relation on the set is an "information content" relation. If S_1 and S_2 are two structures, we write

$$S_1 \leq S_2$$

and read "S_1 is less defined than or equal to S_2."

If P is a partially ordered set (poset), we let $\leq$ (or $\sqsubseteq$) denote the order relation in P, and thus for $p_1, p_2 \in P$ we write

$$p_1 \leq p_2$$

to denote the fact that p_1 is less than or equal to p_2, or p_1 is less defined than or equal to p_2. $\sqcup$("cup") denotes binary or finite least upper bounds, and $\bigsqcup$("mug") denotes least upper bounds of arbitrary sets.

Definition 12 A partially ordered set $(P,\leq)$ is a set P together with a binary relation $\leq$ which is reflexive, transitive and antisymmetric. All posets are assumed to have a minimum element denoted $\perp$("bottom" or undefined) such that

$$\perp \leq p$$

for any $p \in P$. A subset S of P is directed or a Δ-set iff every finite subset of S has an upper bound in S. A function f: $P \to P'$ from a poset P to a poset P' is monotonic iff for all $p_1 \leq p_2$ in P, $f(p_1) \leq f(p_2)$ in P'.

We say that a subset S of P is an ω-set if S is an ω-chain, that is, a countable set such that $s_1 \leq s_2 \leq s_3 \leq \dots$.

Still following the notation of ADJ[2], we say that for $Z = \omega$ or $Z = \Delta$, a function f: $P \to P$ is Z-continuous if f preserves all least upper bounds of Z-sets that exist in P. That is, f is Z-continuous iff

$$f(\sqcup_{i\in I}p_i) = \sqcup_{i\in I}f(p_i)$$

where $\langle p_i \rangle_{i\in I}$ is a Z-set in P and $\sqcup_{i\in I}p_i$ exists. A poset P is Z-complete iff all Z-sets have least upper bounds in P. □

In order to define infinite trees as the appropriate partially ordered set we use the following definition:

Definition 13 For sets A, B let $[A \multimap B]$ be the poset of partial functions from A to B. The order relation is set inclusion on the graphs of the functions considered as subsets of $A \times B$. The least upper bound is set union. If

$$f: A \multimap B$$

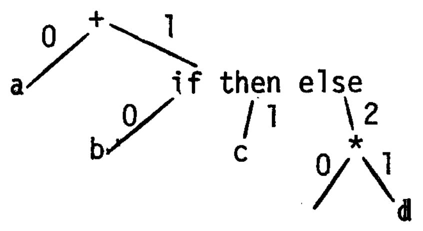
is a partial function, then

$$\mathrm{def}(f) = \{a \mid \langle a,b \rangle \in f\}$$

is the domain of definition of f. If $x \notin \mathrm{def}(f)$, we write $f(x) = \perp$. □

The definition of trees (including infinite trees) as posets is achieved by labelling the branches of each tree by integers on each level of the tree. Then each node of the tree is in the target of a partial function defined over strings of integers. These strings represent the path taken from the root to reach the node.

For example, consider the following tree



This is the partial function t defined by

$\lambda \to +$

$0 \to a$

$1 \to$ ifthenelse

$10 \to b$

$11 \to c$

$12 \to *$

$121 \to d$

(Note that $t(120)$ is undefined).

Every tree is a partial function, but not every partial function from strings of integers to operators is a tree. The formal definition is given in definition 14 below. The sort of the tree is defined as the sort of the operator at the root of the tree. If $t(\lambda)$ is undefined, then the entire tree is undefined. The ordering on trees is defined by set inclusion on partial functions with set union giving upper bounds if they exist; that is if the union is a function satisfying definition 14.

Definition 14 Let ω denote the set of natural numbers. We say that a *Σ-tree of sort* $s \in S$ is a partial function

$$t\colon\ \omega^* \multimap \Sigma$$

such that

(i) if $\lambda \in \mathrm{def}(t)$ then $t(\lambda)$ has sort s.

(ii) If $w \in \omega^*$, $i \in \omega$ and $wi \in def(t)$ then

a) $w \in def(t)$

b) if $t(w)$ has arity $s_1 s_2 \ldots s_n$, then $i < n$ and $t(wi)$ has sort s_{i+1}. □

Condition (i) defines the sort of the tree, while condition (ii) ensures that (a) a node is defined only if its father is defined and (b) the "arguments" to each operator are of the correct sort and if an operator is n-ary, n or fewer arguments are defined in the tree.

Definition 15 Let $CT_{\Sigma,s}$ denote the set of all Σ-trees of sort s. The Σ-algebra CT_Σ is defined as follows: The carrier of CT_Σ is $<CT_{\Sigma,s}>_{s \in S}$ for each $s \in S$. The operators are defined as follows:

(i) For $\sigma \in \Sigma_{\lambda,s}$, $\sigma_{CT} = \{<\lambda,\sigma>\}$

(ii) For $\sigma \in \Sigma_{w,s}$, $w = s_1 s_2 \ldots s_n$ and $t_{s_i} \in CT_{\Sigma,s_i}$, $1 \leq i \leq n$

$\sigma_{CT}(t_{s_1},\ldots,t_{s_n}) = \{<\lambda,\sigma>\} \cup \bigcup_{i<n}\{<iu,\sigma'> | <u,\sigma'> \in t_{i+1}\}$. □

So if $t_{s_1},\ldots,t_{s_n}$ are Σ-trees, so is $\sigma_{CT}(t_{s_1},\ldots,t_{s_n})$ - it is the tree with σ as root.

Definition 16 A Σ-algebra is Z-continuous iff each member of its carrier is strict (has a minimum element $\perp$) and is Z-complete, and if its operations are Z-continuous. A data type is said to be Z-continuous if it is Z-continuous as a many-sorted algebra. A function $f\colon A \to B$ is <u>strict</u> if $f(\bot_A) = \bot_B$. $\square$

The following important result is proven in ADJ [2].

<u>Theorem 6</u> CT_Σ is initial in the class of ω-continuous Σ-algebras with strict Δ-continuous Σ-homomorphisms. $\square$

As before with T_Σ, we let $CT_\Sigma(X_n)$ denote the initial $\Sigma(X_n)$-algebra. An element $x_i \in X_n$ is called a <u>variable</u>.

<u>Theorem 7</u> If A is a Z-continuous Σ-algebra and

$$a\colon\ X_n \to A$$

is an assignment, then

$$\bar{a}\colon\ CT_\Sigma(X_n) \to A$$

is the unique Z-continuous homomorphism determined by making A into a $\Sigma(X_n)$ algebra. $\square$

Again, if $t \in CT_\Sigma(X_n)$, $\text{sort}(t) = s$, $\text{arity}(x_i) = s_i$ for each i, $1 \le i \le n$, then

$$t_A\colon\ A_{s_1} \times A_{s_2} \times \dots \times A_{s_n} \to A_s$$

is defined by

$$t_A(a) = \bar{a}(t).$$

$[A \multimap B]$ is strict and Δ-complete, and in fact CT_Σ is strict and Δ-complete. Furthermore for any $t \in CT_\Sigma$, $t = \sqcup_{n\in\omega} t^{(n)}$ such that $\mathrm{def}(t^{(n)})$ is finite. This result is of fundamental importance since it enables one to discuss properties of infinite objects by discussing the properties for each finite object and then showing that the properties hold in the limit. So for example if f is a continuous function

$$f: \; CT_\Sigma \to CT_\Sigma$$

then $f(t) = \sqcup_i f(t_i)$ by definition of continuity. The operations of CT_Σ are Δ-continuous.

The notion of continuity is extended to congruences in the following definition.

Definition 17 A Σ-congruence q on a Z-continuous Σ-algebra A is said to be Z-continuous for $Z \in \{\Delta, \omega\}$ if whenever $\langle t_i \rangle_{i\in I}$ and $\langle \bar{t}_i \rangle_{i\in I}$ are Z-sets in A such that

$$(t_i, \bar{t}_i) \in q \quad \text{for each } i \in I$$

then

$$(\sqcup_i t_i, \sqcup_i \bar{t}_i) \in q. \qquad \square$$

This ensures that the congruences under consideration are well behaved in some sense. For example, suppose that there is an algebra with one sort S and operators

$$+: S \to S$$

$$\perp: \to S$$

and with the equation $+.\perp = \perp$.

If q is the least congruence generated by this axiom, then q is not continuous and CT_Σ/q will not be continuous. To see this, consider the chain

$$< +^i.\bot >_{i\in\omega}$$

and let $t = \sqcup_i < +^i.\bot >$.

Now $+[\sqcup_i <+^i.\bot>] = [+\sqcup_i +^i\bot]$ by definition of $+_{CT_\Sigma/q}$.

$= [\sqcup_i +^{i+1}\bot]$ by continuity of CT_Σ

$= [\sqcup_i +^i.\bot]$

But $+(\sqcup_i[+^i.\bot]) = +(\sqcup_i[\bot]) = [+\bot] = [\bot]$.

Now since q is the least congruence generated by the equation, we have

$$[\sqcup_i +^{i+1}.\bot] \neq [\bot]$$

and CT_Σ/q is thus not continuous.

We now generalize theorem 3 to Δ-continuous Σ-algebras.

Theorem 8 If A is a Δ-continuous Σ-algebra and R is a relation on A, then there exists a least Δ-continuous Σ-congruence on A containing R, called the Δ-continuous congruence relation generated by R on A.

Proof Let $K(R)$ be the class of all Δ-continuous Σ-congruence relations on A that contain R. $K(R) \neq \phi$ since

$$U = <u_s = A_s \times A_s \mid s \in S>$$

is in $K(R)$, and is Δ-continuous since $(a_1,a_2) \in U$ for any $a_1,a_2 \in A$. Let $\equiv_R = \cap K(R)$. That is

$$(\equiv_R)_s = \cap\{K_s \mid K \in K(R)\} \quad \text{for each } s \in S.$$

It is shown in ADJ [1] that $\equiv_R$ is a Σ-congruence relation. We show that $\equiv_R$ is Δ-continuous. Suppose that $\langle t_i \rangle_{i \in I}$ and $\langle \bar{t}_i \rangle_{i \in I}$ are directed sets in A with

$$a = \sqcup_i a_i, \quad \bar{a} = \sqcup_i \bar{a}_i.$$

Also, suppose that

$$\langle a_i, \bar{a}_i \rangle \in \equiv_R \qquad \text{for each } i \in I$$

Hence

$$\langle a_i, \bar{a}_i \rangle \in K \quad \text{for each } i \in I \text{ and each } K \in \mathcal{K}(R).$$

by definition of $\equiv_R$.

But each $K \in \mathcal{K}(R)$ is Δ-continuous, so

$$\langle a, \bar{a} \rangle \in K$$

for each $K \in \mathcal{K}(R)$, and thus $\langle a, \bar{a} \rangle \in \equiv_R$ as required. $\square$

<u>Definition 18</u> A system of n (<u>regular</u>) <u>equations</u> in CT_Σ is a function

$$e: X_n \to CT_\Sigma(X_n)$$

such that $\text{sort}(x_i) = \text{sort}(e(x_i))$.

For any Δ-continuous Σ-algebra A, if $w = s_1 s_2 \ldots s_n$ and $\text{sort}(x_i) = s_i$, then

$$e_A: A^w \to A^w$$

is the <u>derived operator</u> of e over A defined as

$$e_A(\langle a_{s_1}, a_{s_2}, \ldots, a_{s_n} \rangle) = \langle e(x_1)_A(a_{s_1}), \ldots, e(x_n)_A(a_{s_n}) \rangle$$

$$= \langle \bar{a}_{s_1}(e(x_1)), \bar{a}_{s_2}(e(x_2)), \ldots, \bar{a}_{s_n}(e(x_n)) \rangle.$$

That is, if $a = \langle a_{s_1}, \ldots, a_{s_n} \rangle$ is an n-tuple in a Σ-algebra A, then regarding a as the assignment

$$x_i \mapsto a_{s_i} \qquad 1 \leq i \leq n$$

we define $e_A(a)$ as the n-tuple obtained from the unique homomorphism $\bar{a}$ extended from a and applied to $e(x_1), e(x_2), \ldots, e(x_n)$. □

<u>Theorem 9</u> The system of equations

$$e_A: A^w \to A^w$$

has a minimal fixed point solution denoted $|e_A| \in A^w$ called the <u>solution</u> of e over A.

Furthermore

(i) $|e_A| = \sqcup_{k \in \omega} e_A^k(\perp, \ldots, \perp)$

(ii) $e_A(|e_A|) = |e_A|$

and (iii) For all $a \in A^w$ if $e_A(a) = a$ then $|e_A| \sqsubseteq a$.

(Here $e_A^0(\perp, \ldots, \perp)$ denotes $(\perp, \ldots, \perp)$

$e_A^k(x)$ denotes $e_A(e_A^{k-1}(x))$ for $k \geq \perp$. □

<u>Definition 19</u> Finally, we can define substitution of expressions for variables by considering the case $A = CT_\Sigma(X_m)$ above. Then let $t = \langle t_{s_1}, \ldots, t_{s_p} \rangle \in (CT_\Sigma(X_n))^w$ be a p-tuple where $w = s_1 s_2 \ldots s_p$. This can be considered as a mapping.

$$t: X_p \to CT_\Sigma(X_n)$$

that is $t(x_i) = t_i$. Then if $t' = \langle t'_1, \ldots, t'_n \rangle$ is an n-tuple of m-ary terms in $CT_\Sigma(X_m)$, we write $t \leftarrow t'$ for the substitution of the n-tuple of m-ary terms in $CT_\Sigma(X_m)$ in the p-tuple of n-ary terms in $CT_\Sigma(X_n)$. This is defined as follows

$$t \leftarrow t' = \bar{t}' \circ t$$

where $\circ$ is function composition, and $\bar{t}'$ is the unique Σ-homomorphism extending

$$t' \colon X_n \rightarrow CT_\Sigma(X_m).$$

<u>Theorem 10</u> Substitution is Δ-continuous. i.e.

$$\leftarrow \colon (CT_\Sigma(X_n))^p \times (CT_\Sigma(X_m))^n \rightarrow (CT_\Sigma(X_m))^p$$

is a Δ-continuous function on the (Δ-complete) Cartesian product of Δ-complete sets. □

Chapter 3

Sharing and Circularity Using References

1 Introduction

As outlined in the previous chapter, a data type may be regarded as a many-sorted algebra which is initial in some particular class of algebras. The class of algebras is defined by a set of equations called the specification. Usually, assignment is not treated as an operator of the type, and variables are not treated in the formalism. For example, consider the specification of the type stack.

$$\begin{array}{ll}
\text{pop:} & \text{stack} \to \text{stack} \\
\text{push:} & D \times \text{stack} \to \text{stack} \\
\text{top:} & \text{stack} \to D \\
\text{empty:} & \text{stack} \to \{\text{true, false}\} \\
\Lambda: & \to \text{stack}
\end{array}$$

$$\begin{array}{ll}
\text{pop}(\text{push}(d,s)) & = s \\
\text{top}(\text{push}(d,s)) & = d \\
\text{empty}(\Lambda) & = \underline{\text{true}} \\
\text{empty}(\text{push}(d,s)) & = \underline{\text{false}}
\end{array}$$

(This definition should also define top(Λ) see ADJ[1].)

In an actual programming language there are two possible ways that stack can be implemented; either the operators are functional, or they work by side effect. In the first case, for example, we may write

```
x ← push(a,push(b,push(c,d)))
y ← pop(x)  etc.
```

while in the second case we might write

```
push(a,x)
push(b,x)
push(a,x)
y ← x
pop(y)
```

However, in spite of the so-called "specification" of the type given above, some questions still remain unanswered.[†] For example, what is the semantics of

```
y ← x?
```

Does pop(y) affect x? Is it possible to formulate proof rules for the various statements?

We show in this chapter that a more complete specification of a type can be given by including assignment as an operator of the type, and by adding to the carrier of the type a set called struct which,

[†] The difficulty of treating errors is not discussed here, but is discussed in ADJ[1] and Goguen [16]. A more practical approach is taken in Guttag et al [17] and Tompa [40].

intuitively, captures the interrelationships of variables of the type. Also, variables themselves are added, either as an extra set V in the carrier, or, more elegantly, as operators of the type, either nullary or unary. For example, we might say that the set of sorts for stack is

{struct, stack, D}

and the operators are (say)

push: $D \times \text{stack} \times \text{struct} \to \text{struct}$

x: $\text{struct} \to \text{stack}$ for each $x \in V$.

The intention that push works by "side effect" can be described axiomatically, since

push(d, x(σ), σ)

is regarded as yielding a new structure from a given structure σ. It becomes possible by including variables and the set struct to define types which allow sharing or aliasing. This is illustrated through the use of the general type linked-list, which includes operators similar to REPLACD and REPLACA of "impure" LISP [32]. The principles are, however, easily applied to other types, such as stack.

The most important aspect of this formulation is that it yields our abstract model of the type. The type is not modeled by any particular operational device such as graphs or location-value functions, but, in the spirit of abstract data types, only the abstract properties of the type are presented. It then becomes possible to use a very simple proof rule for programs using the type. Use of this rule is illustrated briefly in Chapter 5. Because of the abstract nature of the axioms, we believe that it is easier to argue correctness. Existing verification methods for this type are all based on operational models, and hence need more detailed, but non-essential, reasoning in verification.

2 Referencing and Verification

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 ← cons(a,cons(b,c))
2.        y ← tℓ x
3.        tℓ y ← z
```

where ← 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[32]) 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: $x = cons(a,cons(b,cons(d,e)))$ & $y = cons(b,cons(d,e))$ &
$z = cons(d,e)$,

then the sharing between x, y and z is not explicitly expressed. For example, the following assignments also satisfy the above assertion:

1. $x \leftarrow cons(a,cons(b,cons(d,e)))$
2. $y \leftarrow cons(b,cons(d,e))$
3. $z \leftarrow cons(d,e)$

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 after

$x \leftarrow cons(a,b)$

$t\ell\ x \leftarrow x$

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 sort of operational model of lists. (See for example Oppen and Cook [31]). 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. Hence suppose that σ is the name of a data structure. We view an assignment of the form

$$p \leftarrow e$$

where p is some expression denoting a component of σ, as in fact being the assignment

$$\sigma' \leftarrow u(\sigma,p,e).$$

Here $U(\sigma,p,e)$ is a function whose value is a structure identical to σ except p is changed to e. Hence the proof rule we need is

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

where $Q^{\sigma}_{U(\sigma,p,e)}$ is Q with each occurrence of σ replaced by $U(\sigma,p,e)$. This view is similar to the view that an array assignment such as

$$A[i] \leftarrow e$$

changes the entire array A. (See Hoare and Wirth [22]. Compare this rule with the "simple" assignment rule in Hoare [19].)

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

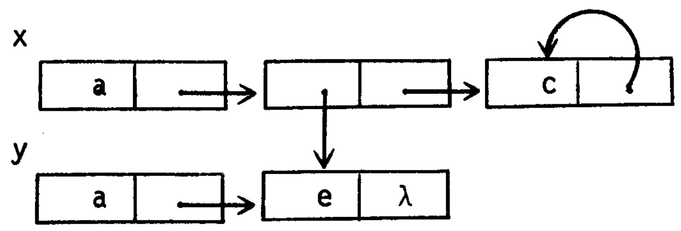
$$Q^{\sigma}_{\sigma'}\{p \leftarrow e\}\ Q$$

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. Rather than being a drawback, it will be seen that this is advantageous since, together with the treatment of referencing discussed below, it allows properties of a structure to be simply expressed. The remaining idea needed for the proof rule above to work is to describe more precisely the concept of a cell and reference.

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 cell may change after an assignment, and hence 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 [38] 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 [6].)

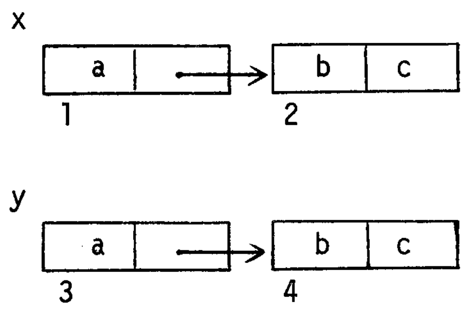
The type to be described has essentially the same semantics as the LISP type (See McCarthy et al [32]), but uses Algol 60 syntax (as for example in Burstall [9]). Roughly speaking an element of the type is called a list structure (or structure) and it consists of a finite set of cells or nodes. Each cell has two components called the head and tail of the cell. The head and tail of a cell each are 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 <u>atoms</u>, with λ being a distinguished element called <u>null</u>.

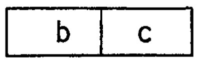
The operators of the type include the following three operators: hd and *tℓ* which take as argument a cell and return the head and tail of the cell respectively, and cons which takes two arguments (atoms or cells) and returns as its value the description of a cell containing in its head and tail its two arguments. There are also two types of assignment operators in the type. <u>Simple</u> assignment, written in the form $x \leftarrow c$, associates a variable x with a cell c. A more general assignment operator changes the head or tail of a cell. This assignment is written in the form hd $c_1 \leftarrow c_2$ (or *tℓ* $c_1 \leftarrow c_2$) and the intended semantics is that the head (or tail) of cell c_1 is changed to refer to c_2. hd and *tℓ* correspond to CAR and CDR in LISP, while ← corresponds to CSET. hd $c_1 \leftarrow c_2$ corresponds to REPLACA(c_1,c_2) and *tℓ* $c_1 \leftarrow c_2$ 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.

Consider the list structure:

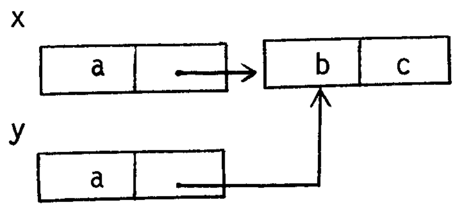


(The boxes have been numbered arbitrarily.)

We could write the descriptions of boxes 2 and 4 as follows:



If no distinction is made between references and the content of a cell, then it is not possible to distinguish between the above structure and the structure



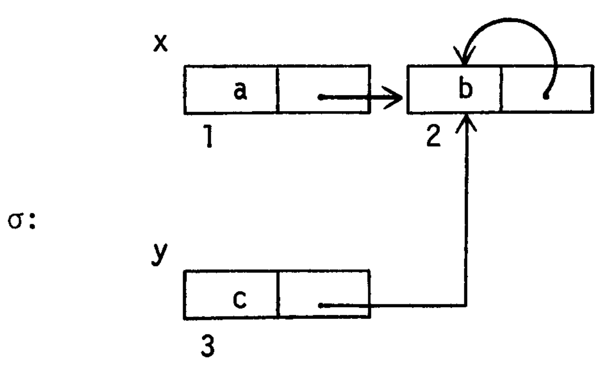
However, these two structures are clearly different as can be seen by considering the effect on the structures of the assignment

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

which effects only cells accessible from x in the first case, but affects cells accessible from either x or y in the second case.

3 A Quotient Approach to References

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, $t\ell$, x,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

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

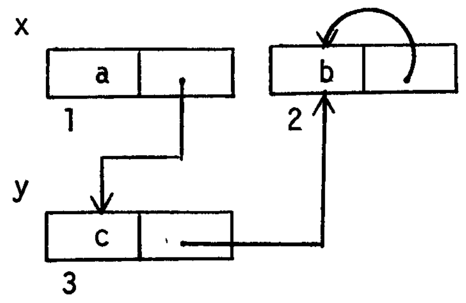
is a sequence of structures where σ_i is obtained from σ_{i-1}, $i > 0$, by an update. Then references in σ_j can be related to references in σ_k, $j,k \in \{0 \ldots n\}$, by relating cells in the updated structure to cells in the earlier structure. Conceptually, the new set of cells is obtained from the old by changing the contents of 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

$$t\ell\ x \leftarrow y.$$

That is, the updated structure is

σ'



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 references in 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 2) or $t\ell(x(\sigma))(\sigma')$ (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

$$\text{cons}(d_1,d_2)$$

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

$$\text{hd}(\text{cons}(r_1,r_2))(\sigma)$$

where r_1 is a reference. The subexpression

$$\text{cons}(r_1,r_2)$$

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

$$\text{hd}(\text{cons}(r_1,r_2))(\sigma) = r_1$$

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 $\text{hd}(r_1)(\sigma)$ and $\text{hd}(r_2)(\sigma)$ 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

$$\text{cons}(c_1,c_1') = \text{cons}(c_2,c_2')$$

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.

4 The Type Linked List

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

ϕ:	$\to$ Struct	(the empty structure)
hd, $t\ell$:	Cell $\times$ Struct $\to$ Cell	
cons:	Cell $\times$ Cell $\to$ Cell	
x:	Struct $\to$ Cell	for each $x \in V$
x:	$\to$ V	for each $x \in V$
Error:	$\to$ Cell	
a:	$\to$ Cell	for each $a \in$ Atom
$\leftarrow$:	V $\times$ Cell $\times$ Struct $\to$ Struct	
$\leftarrow_{hd}$:	Cell $\times$ Cell $\times$ Struct $\to$ Struct	
$\leftarrow_{t\ell}$:	Cell $\times$ Cell $\times$ Struct $\to$ Struct	
$\approx$:	Cell $\times$ Cell $\to$ Bool	
$\downarrow$:	Cell $\times$ Struct $\to$ Cell	(dereferencing)

We also assume that the sort Bool with operators

<u>true</u>, <u>false</u>: $\to$ Bool

if then else: Bool $\times$ t $\times$ t $\to$ t   for each sort t

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

if then else (<u>true</u>, t_1,t_2) = t_1

if then else (<u>false</u>,t_1,t_2) = t_2.

An alternative approach is to include the sets <u>reference</u> and <u>description</u> in the carrier of the type, where elements of reference are references and elements of description are descriptions. However to simplify the specification of the type, the set of sort cell is considered to include both references and descriptions. A given element of cell will either be a reference or a description. We define an operator

isref: Cell $\to$ Bool

which returns true if and only if its argument is a reference.

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

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

Error is a constant of sort Cell which is used to denote an erroneous reference.

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: Struct $\rightarrow$ Cell 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, consider the functional description of hd, $t\ell$. Suppose $c \in$ Cell. Then hd(c) depends on some particular structure σ. We write hd(c)(σ) and tl(c)(σ) to denote hd(c,σ) and tl(c,σ) respectively. Thus $\text{hd}(t\ell(x(\sigma))(\sigma))(\sigma) \in$ List for some $\sigma \in$ Struct. The elements of cell which are of the form $\delta_1(\delta_2(\ldots(\delta_n(x(\sigma_{n+1}))(\sigma_n)\ldots)(\sigma_1)$ where $\delta_i \in \{\text{hd}, t\ell\}$, $1 \le i \le n$, $\sigma_i \in$ Struct, $1 \le i \le n+1$, will be called <u>references</u>. If $\sigma_i = \sigma$ for each i, $1 \le i \le n+1$, the the element will be called a <u>reference in σ</u>, and we will write $\delta_1\delta_2\delta_3\ldots\delta_n\ x(\sigma)$. 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 47) may have been derived from

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

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

$$\leftarrow(\delta p_1, p_2, \sigma)$$

to denote

$$\leftarrow_\delta(p_1, p_2, \sigma)$$

where $\delta \in \{hd, t\ell\}$.
For example, we write $\leftarrow(t\ell\ x(\sigma), cons(a,b), \sigma)$ to denote $\leftarrow_{t\ell}(x(\sigma), cons(a,b), \sigma)$.

(ii) In an assignment of the form

$$\leftarrow(x, c_2, \sigma) \quad\text{or}\quad \leftarrow(\delta c_1, c_2, \sigma),$$

if all structures occurring in the expressions c_1 and c_2 are σ, then σ is omitted from c_1 and c_2.
For example, we write

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

to denote

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

(iii) As mentioned above, 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

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

If the structures are different, we use a different notation. For example, if

$\vec{\sigma} = \langle\sigma_1,\sigma_2,\sigma_3\rangle$

we will write

$\mathrm{hd}\ t\ell\ x(\vec{\sigma})$

to denote

$\mathrm{hd}(t\ell(x(\sigma_1))(\sigma_2))(\sigma_3)$. □

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

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

(ii) $\leftarrow(\delta c_1,c_2,\sigma)$ updates σ to a new structure σ' which is identical to σ except that the contents of cell c_1 are changed. If $\delta = \mathrm{hd}$, then the head of c_1 is changed to reference c_2. If $\delta = t\ell$, then the tail of c_1 is changed to reference c_2. We do not allow c_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

$\leftarrow(\delta c_1,\ \mathrm{cons}(c_2,c_3),\sigma)$

(which is not allowed) is equivalent to

$$\leftarrow(\delta c_1,\ z,\ \leftarrow(z, \mathrm{cons}(c_2,c_3),\sigma)).$$

<u>Definition 1</u> The Type Axioms.

Let Path = {hd, $t\ell$}* · V (that is, expressions of the form qx where $q \in \{hd,t\ell\}^*$, $x \in V$). Let $c_1,c_2,c \in$ ell, $q,\bar{q} \in \{hd,t\ell\}^*$, $p,p_1,p_2,p_3 \in$ Path, $\delta \in \{hd,t\ell\}$ and $\sigma,\sigma',\sigma_1,\sigma_1',\sigma_2,\sigma_2' \in$ Struct, $a \in$ Atom, x, $\bar{x}$, $y \in V$.

1. $\delta(p(\sigma))(\phi) = \text{Error}$

2. $q(a)(\sigma) = \text{Error}$

3. $hd(\mathrm{cons}(c_1,c_2))(\sigma) = c_1$

4. $t\ell(\mathrm{cons}(c_1,c_2))(\sigma) = c_2$

5. $x(\leftarrow(\delta p, c,\sigma)) = x(\sigma)$

6. $$qx(\leftarrow(y,c,\sigma)) = \begin{cases} qc(\sigma) & \text{if } x == y \text{ and isref}(qc(\sigma)) \\ qx(\sigma) & \text{if } x \neq= y \end{cases}$$

7. $$\delta(qx(\sigma'))(\leftarrow(y,c,\sigma)) = \begin{cases} \delta(qx(\sigma'))(\sigma) & \text{if } \sigma' \neq= \leftarrow(y,c,\sigma) \text{ and} \\ & \quad \mathrm{Df}(\leftarrow(y,c,\sigma),\ \sigma'), \\ \text{Error} & \text{if } \mathrm{Df}(\sigma',\leftarrow(y,c,\sigma)) \text{ and} \\ & \quad \sigma' == \leftarrow(x,c_1,\bar{\sigma}') \text{ and} \\ & \quad \text{isref}(qc_1(\bar{\sigma}')). \end{cases}$$

where

$$\mathrm{Df}(\sigma_1,\sigma_2) = \begin{cases} \underline{\text{false}} & \text{if } \sigma_1 == \phi \\ \underline{\text{true}} & \text{if } \sigma_1 == \sigma_2 \\ \mathrm{Df}(\bar{\sigma}_1,\sigma_2) & \text{otherwise, where } \sigma_1 == \leftarrow(p,c,\bar{\sigma}_1) \end{cases}$$

8. a) $hd(p_1(\sigma_1))(\leftarrow(t\ell p_2,p_3,\sigma)) = hd(p_1(\sigma_1))(\sigma)$

b) $t\ell(p_1(\sigma_1))(\leftarrow(hdp_2,p_3,\sigma)) = t\ell(p_1(\sigma_1))(\sigma)$

9\.
$$\delta(p_1(\sigma_1))(\leftarrow(\delta p_2,p_3,\sigma)) = \begin{cases} p_3(\sigma) & \text{if } p_1(\sigma_1) \simeq p_2(\sigma) \\ \delta(p_1(\sigma_1))(\sigma) & \text{otherwise} \end{cases}$$

10\.
$$\simeq(qx(\sigma_1'),\bar{q}\bar{x}(\sigma_2')) = \begin{cases} \underline{\text{true}} \text{ if } \sigma_1' == \sigma_2' \ \& \ \text{isref}(qc_1(\sigma_1)) \ \& \\ \quad \text{isref}(\bar{q}c_2(\sigma_2)) \ \& \ qx == \bar{q}\bar{x} \\ \underline{\text{false}} \text{ if } \sigma_1' == \sigma_2' \ \& \ \text{isref}(qc_1(\sigma_1)) \ \& \\ \quad \text{isref}(\bar{q}c_2(\sigma_2)) \ \& \ qx \neq \bar{q}\bar{x} \end{cases}$$

where $\sigma_1' = \leftarrow(x,c_1,\sigma_1), \sigma_2' = \leftarrow(\bar{x},c_2,\sigma_2)$.

11\.
$$\text{isref}(c) = \begin{cases} \underline{\text{false}} \text{ if } c == \text{Error},\ c \in \text{Atom or} \\ \qquad c == \text{cons}(c_1,c_2) \\ \underline{\text{true}} \text{ if } c == p(\sigma). \end{cases}$$ □

We have used the symbol "==" to denote syntactic equality between terms. This equality can be defined equationally on terms in the obvious way.

These axioms may be read intuitively as follows:

1,2: These references are erroneous.

3,4: The "usual" list axioms.

5: A variable references the same cell after an assignment that is not a simple assignment.

6: If σ is updated by a simple assignment to a variable, then references involving that variable may be affected by the assignment. If the assignment is say $\sigma' = \leftarrow(x,c,\sigma)$ and c is a description, then $qx(\sigma')$ will be equivalent to a reference in σ if $qc(\sigma')$ is a reference.

7: Df(σ_1,σ_2) is true if σ_1 is derived from σ_2 by a sequence of assignments, i.e.

$\sigma_1 = \leftarrow(p_1,c_1,\leftarrow(p_2,c_2,\ldots\leftarrow(p_k,c_k,\sigma_2)))\ldots)$.

If $qx(\sigma')$ is a reference, then $\delta(qx(\sigma'))(\sigma_1)$ is erroneous if σ' is derived from σ_1, but in some sense $qx(\sigma')$ references a cell created "after" σ_1. (That is, the cell is undefined in σ_1.) If σ_1 is derived from σ', then a simple assignment cannot affect the reference $\delta(qx(\sigma'))(\leftarrow(y,c,\sigma))$ unless $\sigma' = \leftarrow(y,c,\sigma)$.

8: Assignment to the head of a cell cannot affect the tail of any cell, and vice-versa.

9: The head or tail of a cell referenced by $p_1(\sigma')$ is changed if the head or tail of that cell is assigned a reference. No other cell is changed.

10: $\simeq$ is a predicate that tests for equality of two references. This axiom implicitly uses the result that each reference has a canonical form $qx(\leftarrow(x,c,\sigma))$ where $\sim$isref$(qc(\sigma))$. This result is shown in the next section.

11: isref tests whether an element $c \in$ Cell is a reference. Note that by omission we have disallowed assignments of the forms

a) $\leftarrow(\text{cons}(c_1,c_2),c,\sigma)$ and

b) $\leftarrow(\delta\ p,\text{cons}(c_1,c_2),\sigma)$.

Case (a) is omitted because $\mathrm{cons}(c_1,c_2)$ does not identify a cell. Case (b) is omitted because it severely complicates the dereferencing mechanism.

We have not treated errors here, but following Goguen [16] we might write the following error axioms:

12: $hd(a) = \mathrm{Error}_C$ if $a \in \mathrm{Atom}$

13: $t\ell(a) = \mathrm{Error}_C$ if $a \in \mathrm{Atom}$

14: $\leftarrow(\mathrm{cons}(c_1,c_2),c,\sigma) = \mathrm{Error}_S$

15: $\leftarrow(\delta\, p,\mathrm{cons}(c_1,c_2),\sigma) = \mathrm{Error}_S$

where Error_C and Error_S are distinguished elements of type Cell and Struct respectively.

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 hand sides of the axioms, where this is necessary. The symbol '$\equiv$' is used to denote the congruence relation defined by these axioms.

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

$$\begin{array}{l} p_1 \leftarrow c_1;\ \sigma_0 \\ p_2 \leftarrow c_2;\ \sigma_1 \\ \quad\vdots \\ p_n \leftarrow c_n;\ \sigma_{n-1} \end{array}$$

to denote the fact that $\sigma_0 = \leftarrow(p_1,c_1,\phi)$ and $\sigma_i = \leftarrow(p_{i+1},c_{i+1},\sigma_{i-1})$ for $1 \le i \le n$.

Example 1

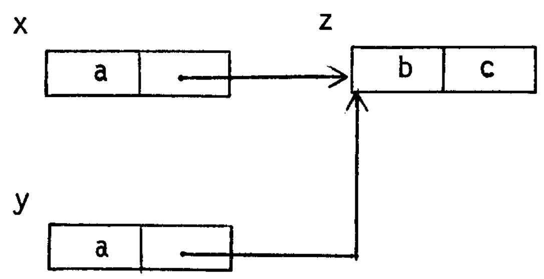
1. Consider the assignments

$x \leftarrow \text{cons}(a,\text{cons}(b,c));$ 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 $t\ell\ y(\sigma_2)$. By axiom 6

$t\ell\ y(\sigma_2) \equiv z(\sigma_1)$

$\equiv t\ell\ x(\sigma_0)$ by axiom 6 again.

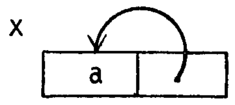
Similarly, $t\ell\ x(\sigma_2) \equiv t\ell\ x(\sigma_1)$ by axiom 6

$\equiv t\ell\ x(\sigma_0)$ by axiom 6.

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

$t\ell\ x \leftarrow x;$ σ_1

Here σ_1 is represented by



$t\ell\ t\ell\ t\ell\ x(\sigma_1) = t\ell\ (t\ell(t\ell(x(\sigma_1))(\sigma_1))(\sigma_1))(\sigma_1)$

by notational convention.

Now $\qquad x(\sigma_1) \equiv x(\sigma_0)$ $\qquad$ by axiom 5

and $\qquad tl(x(\sigma_0))(\sigma_1) \equiv x(\sigma_0)$ $\qquad$ by axiom 9, since $x(\sigma_0) \simeq x(\sigma_0)$.

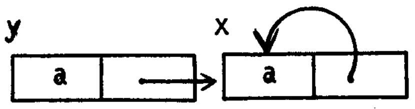
Hence (by symmetry)

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

3. The above example illustrated the treatment of circularity. Circularity may also be introduced indirectly.

$$x \leftarrow cons(a,cons(b,c));\ \sigma_0$$
$$y \leftarrow cons(a,x);\qquad \sigma_1$$
$$tltl\ y \leftarrow x;\qquad \sigma_2$$

σ_2 may be represented by



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

$x(\sigma_2) \equiv x(\sigma_1) \equiv x(\sigma_0)$ $\qquad$ by axiom 5 and 6.

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

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

$tl\ y(\sigma_1) \equiv x(\sigma_0)$ $\qquad$ by axiom 6

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

Again by symmetry, therefore,

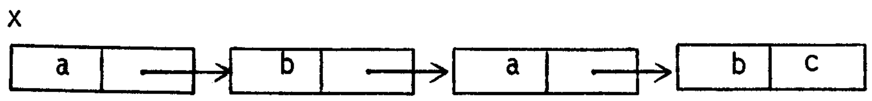
$tl\ tl\ tl\ x(\sigma_2) \equiv x(\sigma_0)$.

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 $t\ell\ x(\sigma_1)$ cannot be expressed in terms of σ_0, but

$t\ell\ t\ell\ x(\sigma_1) \equiv x(\sigma_0)$ by axiom 6. □

These examples illustrate that references of the form

$px(\vec{\sigma})$

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

$qy(\sigma')$

such that σ' is of the form

(y,c,σ'')

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. It is possible to define a dereferencing operator (discussed in section 2) as follows:

<u>Definition 2</u> Dereferencing

$\downarrow$: Cell $\times$ Struct $\rightarrow$ Cell

16: $\downarrow(cons(c_1,c_2),\sigma) = cons(\downarrow(c_1,\sigma), \downarrow(c_2,\sigma))$

17: $\downarrow(a,\sigma) = a$ for all $a \in$ Atom

18: $\downarrow(px(\leftarrow(x,c,\sigma)),\sigma')=$

```
if atom (pc(σ)) then
      ↓(pc(σ),σ')
else
      cons(↓(hd px(σ'),σ'), ↓(tℓ px(σ'),σ')). □
```

Example 2

1.
```
         x ← cons(a,cons(b,c));   σ0
         y ← cons(a,cons(d,e));   σ1
      tℓ x ← y;                   σ2
      hd x ← y;                   σ3
```

$\downarrow(x(\sigma_3),\sigma_3) \equiv \downarrow(x(\sigma_0),\sigma_3)$

$\equiv \text{cons}(\downarrow(\text{hd } x(\sigma_3),\sigma_3),\ \downarrow(t\ell\ x(\sigma_3),\sigma_3))$ —(*)

Now

$\downarrow(\text{hd } x(\sigma_3),\sigma_3) \equiv \downarrow(y(\sigma_1),\sigma_3)$ by reducing hd $x(\sigma_3)$

$\equiv \text{cons}(\text{ hd } y(\sigma_3),\ t\ell\ y(\sigma_3))$

$\equiv \text{cons}(a,\text{cons}(d,e))$

and similarly we can show that

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

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

2.
```
         x ← cons(a,b);     σ0
      tℓ x ← x;             σ1
```

$\downarrow(x(\sigma_1),\sigma_1) \equiv \downarrow(x(\sigma_0),\sigma_1)$

$\equiv \text{cons}(\downarrow(\text{hd } x(\sigma_1),\sigma_1),\ \downarrow(t\ell\ x(\sigma_1),\sigma_1))$

$$\begin{aligned}&\equiv \text{cons}(\downarrow(\text{hd } x(\sigma_0),\sigma_1),\ \downarrow(x(\sigma_0),\sigma_1))\\ &\equiv \text{cons}(a,\downarrow(x(\sigma_0),\sigma_1))\\ &\equiv \text{cons}(a,\text{cons}(a,\downarrow(x(\sigma_0),\sigma_1))\ \text{ etc.}\end{aligned}\qquad \Box$$

5 A Canonical Term Algebra for Linked Lists

We construct in this section a model C for linked list, which is a canonical term algebra. This model has the property that

$$C \cong T_\Sigma/q$$

and is also a subalgebra of T_Σ. (That is, the elements of T_Σ are terms). If an existing model g say is claimed to be equivalent to T_Σ/q, this claim can be verified by showing that

$$g \cong T_\Sigma/q.$$

In practise, this is most easily accomplished by showing that

$$g \cong C.$$

The canonical model also gives some insight into the "abstract" model defined by the axioms. For example references in C turn out to be expressions of the form

$$_qx(\leftarrow(x,c,\sigma))$$

where $_qc(\sigma)$ is not a reference. Intuitively, this reference is the path to some cell via x, where the cell was "created" by the assignment

$$\leftarrow(x,c,\sigma).$$

Because C is canonical, every reference can be represented by such a <u>canonical reference</u>, in the sense that it can be shown to be equivalent to the canonical reference.

Recall from Chapter 2 that a Σ-algebra C is a canonical Σ-term algebra if

$$C_s \subseteq T_{\Sigma,s} \qquad \text{for each } s \in S$$

and $\sigma(t_1\ldots t_n) \in C \Rightarrow t_i \in C$ and $\sigma_C(t_1,\ldots,t_n) = \sigma(t_1\ldots t_n)$.

Furthermore, if C is a canonical Σ-term algebra, and q is the congruence on T_Σ generated by a set of equations ε, then

$$C \cong T_\Sigma/q$$

iff 1) C satisfies ε

and 2) for each $\sigma \in \Sigma_{s_1 s_2 \ldots s_n, s}$ and $t_i \in C_{s_i}$

$$\sigma(t_1\ldots t_n) \equiv_\varepsilon \sigma_C(t_1,\ldots,t_n)$$

Recall also that the type in question has operators

$\phi\colon \to \text{Struct}$

$a\colon \to \text{Cell}$ for each $a \in \text{Atom}$

$x\colon \text{Struct} \to \text{Cell}$ for each $x \in V$

$hd,\ t\ell\colon \text{Cell} \to (\text{Struct} \to \text{Cell})$

$$\text{cons}: \text{Cell} \times \text{Cell} \to \text{Cell}$$
$$\leftarrow: V \times \text{Cell} \times \text{Struct} \to \text{Struct}$$
$$\leftarrow_{hd}: \text{Cell} \times \text{Cell} \times \text{Struct} \to \text{Struct}$$
$$\leftarrow_{t\ell}: \text{Cell} \times \text{Cell} \times \text{Struct} \to \text{Struct}$$

We construct a subalgebra of T_Σ by considering only terms which are in some sense reduced. Descriptions are reduced by "eliminating" hd and $t\ell$ where this is possible. Reduced references are references expressed in terms of the structure in which the referenced cell was created. Reduced structures are structures expressed in terms of reduced cells.

<u>Definition 3</u>

1. A term t of sort cell is said to be <u>a reduced cell</u> if

   i) t is an atom;

   or ii) It is of the form $\text{cons}(t_1,t_2)$ and t_1 and t_2 are reduced;

   or iii) It is of the form

   $px(\leftarrow(x,t,\sigma))$

   where t is reduced, $p \in \{hd,t\ell\}^*$, σ is a reduced structure (defined below) and $pt(\sigma)$ is not a reference.

   or iv) It is of the form $p_1(p_2x(\sigma))(\vec{\sigma})$ where $\vec{\sigma} = <\sigma_0,\sigma_1,\ldots,\sigma_n>$, $|p_1| = n$ and σ,σ_0 are unrelated. That is, $\sim Df(\sigma,\sigma_0)$ and $\sim Df(\sigma_0,\sigma)$ and there is no σ' such that $Df(\sigma_0,\sigma')$, $Df(\sigma,\sigma')$.

2. A term σ of sort Struct is said to be a reduced structure if

   i) $\sigma = \phi$;

   or ii) $\sigma = \leftarrow(x,c,\sigma')$ where c is a reduced cell and σ' is a reduced structure

   or iii) $\sigma = \leftarrow_\delta(c_1,c_2,\sigma')$ where c_1,c_2 are reduced cells and σ' is a reduced structure.

No other terms are reduced. If the sort of the term can be determined by context, then we will simply say that the term is reduced. □

The carrier of the canonical term algebra to be defined will consist of the set of variables V and the sets of reduced terms of sort Struct and Cell, augmented with the distinguished term Error.

<u>Definition 4</u> Let Σ be as given above. We define a Σ-algebra C as follows

1. The carrier is $<V, C_{Cell}, C_{Struct}>$

   where $C_{Cell} = \{c \mid c \in T_{\Sigma,Cell}$ and c is reduced$\} \cup \{Error\}$.

   $C_{Struct} = \{\sigma \mid \sigma \in T_{\Sigma,Struct}$ and σ is reduced$\}$.

2. The operators are defined as follows:

   i) $\phi_C = \phi$

   ii) $a_C = a$; $Error_C = Error$

   iii)
$$x_C(\sigma) = \begin{cases} \text{Error if } \sigma = \phi \\ x(\sigma) \text{ if } \sigma = \leftarrow(x,c,\bar{\sigma}) \text{ and } c \text{ is a description.} \\ x_C(\bar{\sigma}) \text{ if } x \neq y \text{ where } \sigma = \leftarrow(y,c,\bar{\sigma}), \text{ or} \\ \quad\quad \text{if } \sigma = \leftarrow(\delta c_1,c_2,\bar{\sigma}). \end{cases}$$

iv) $\delta_C(\text{cons}(c_1,c_2))(\sigma) = \begin{cases} c_1 & \text{if } \delta = \text{hd} \\ c_2 & \text{if } \delta = t\ell. \end{cases}$

v) $\delta_C(a)(\sigma) = \text{Error}$

vi) $\delta_C(px(\sigma_1))(\sigma_2) = \begin{cases} \text{Error if } \sigma_2 = \phi \text{ or if } \sigma_1 \text{ is derived from } \sigma_2. \\ \delta_C(px(\sigma_1))(\bar{\sigma}_2) \text{ if } \sigma_1 = \sigma_2 = \leftarrow(y,c,\bar{\sigma}_2) \\ \quad \text{and } x \neq y \text{ or if } \sigma_1 \neq \sigma_2, \\ \quad \sigma_2 = \leftarrow(y,c,\bar{\sigma}_2) \text{ and } \sigma_2 \text{ is derived} \\ \quad \text{from } \sigma_1. \\ \delta_C p_C c(\sigma) \text{ if } \sigma_1 = \sigma_2 = \leftarrow(x,c,\sigma), \text{ and} \\ \quad \text{isref}(\delta pc(\sigma_2)). \\ \delta_C px(\sigma) \text{ if } \sigma_1 = \sigma_2 = \leftarrow(x,c,\sigma), \text{ and} \\ \quad \sim\text{isref}(\delta pc(\sigma)). \\ \delta_C(px(\sigma_1))(\sigma) \text{ if } \sigma_2 = \leftarrow(\delta' p_2, p_3, \sigma), \\ \quad \text{and } (\delta' \neq \delta \text{ or } px(\sigma_1) \neq p_2(\sigma)). \\ p_3(\sigma) \text{ if } \sigma_2 = \leftarrow(\delta p_2, p_3, \sigma) \text{ and} \\ \quad p_1 x(\sigma_1) = p_2(\sigma). \\ \delta(px(\sigma_1))(\sigma_2) \text{ if } \sigma_1, \sigma_2 \text{ are unrelated.} \end{cases}$

vii) $\delta_C(p_1(p_2x(\sigma_1))(\vec{\sigma}))(\sigma_2) = \delta(p_1(p_2x(\sigma'))(\vec{\sigma}))(\sigma_2)$ if $\vec{\sigma} = \langle\sigma_0,\sigma_1,\ldots,\sigma_n\rangle$ is not empty.

viii) $\text{cons}_C(c_1,c_2) = \text{cons}(c_1,c_2)$

ix) $\approx_C(c_1,c_2) = \begin{cases} \underline{\text{true}} & \text{if } c_1 = c_2 \\ \underline{\text{false}} & \text{otherwise} \end{cases}$

x) $\leftarrow_C(x,c,\sigma) = \leftarrow(x,c,\sigma)$

xi) $\leftarrow_C(\delta p_1, p_2, \sigma) \qquad = \leftarrow(\delta p_1, p_2, \sigma).$ $\square$

Note that if $p \in \{\text{hd}, t\ell\}^*$, say $p = \delta_1\delta_2\ldots\delta_n$, then $p_C(c)(\sigma)$ denotes $\delta_{1_C}(\delta_{2_C}(\ldots(\delta_{n_C}(c)(\sigma))(\sigma))\ldots)(\sigma)$.

The intention of these operators is, in each case, that given reduced terms, they return reduced terms. It is easily seen that this model $\mathcal{C}$ does satisfy the type axioms because of the close correspondence between the axioms and the definitions of the operators. The only equation which is not easily seen to satisfy the axioms is

$$\delta_C(px(\sigma))(\sigma) = \delta_C p_C c(\bar{\sigma})$$
$$\text{where } \sigma = \leftarrow(x, c, \bar{\sigma}) \text{ and isref}(\delta pc(\sigma)).$$

However note that the axioms represent a schema of axioms, and in particular axiom 6 represents a schema of axioms, one for each $q \in \{\text{hd}, t\ell\}^*$. From axiom 6 we get that if isref(δpc) then

$$qx(\leftarrow(x, c, \sigma)) = qc(\sigma) \quad \text{where} \quad q = \delta p.$$

Hence we must show that for every x, c, σ

$$q_C x_C(\leftarrow(x, c, \sigma)) = q_C(c)(\sigma).$$

But this is exactly the definition of δ_C in this case, and hence the result follows.

We now show that C is canonical.

Lemma 1 C is a canonical term Σ-algebra.

Proof We must show that for each $\sigma \in \Sigma$, if $\sigma(t_1,\dots,t_n) \in C$ then $t_i \in C$ for $1 \le i \le n$ and $\sigma_C(t_1,\dots,t_n) = \sigma(t_1,\dots,t_n)$.

i) $\phi_C = \phi$ from definition (i) of C

ii) for each $a \in$ Atom, $a_C = a$. by definition (ii) of C

iii) If $x(\sigma) \in C$ then $\sigma = \leftarrow(x,c,\bar{\sigma})$ and c is a description. Hence
$x_C(\sigma) = x(\sigma)$ by definition of x_C, as required.

iv) Suppose that $\delta px(\sigma) \in C$. Then $\sigma = \leftarrow(x,c,\bar{\sigma})$ and $\sim$isref($\delta pc(\sigma)$).
Hence $\sim$isref($pc(\sigma)$), so $px(\sigma) \in C$. Furthermore
$\delta_C(px(\sigma))(\sigma) = \delta px(\sigma)$ by definition of δ_C.

v) If cons$(c_1,c_2) \in C$ by definition $c_1,c_2 \in C$. Also,
by definition of cons$_C$,
cons$_C(c_1,c_2)$ = cons(c_1,c_2).

vi) If $\leftarrow(x,c,\sigma) \in C$, then $\sigma \in C$, $c \in C$ by definition. Also,
by definition of $\leftarrow_C$,
$\leftarrow_C(x,c,\sigma) = \leftarrow(x,c,\sigma)$.
Similarly, if $\leftarrow(\delta c_1,c_2,\sigma) \in C$ then $c_1,c_2,\sigma \in C$ and
$\leftarrow_C(\delta c_1,c_2,\sigma) = \leftarrow(\delta c_1,c_2,\sigma)$.

vii) If $\delta(p_1(p_2x(\sigma_1))(\vec{\sigma}))(\sigma) \in C$, then σ_1 and σ_0 are unrelated, where $\vec{\sigma} = <\sigma_0,\sigma_1,\dots,\sigma_n>$, $n \ge 0$, or $p_1 = \lambda$ and σ_1,σ are unrelated.
Then $p_2x(\sigma_1) \in C$, and $p_1(p_2x(\sigma_1))(\vec{\sigma}) \in C$. Finally, by definition of δ_C
$\delta_C(p_1(p_2x(\sigma_1))(\vec{\sigma}))(\sigma) = \delta(p_1(p_2x(\sigma_1))(\vec{\sigma}))(\sigma)$. □

We now show that C characterizes the type defined by the list axioms by showing that each term in T_Σ of the form $\sigma(t_1,\ldots,t_n)$ such that $t_1,\ldots,t_n \in C$ is equivalent to $\sigma_C(t_1,\ldots,t_n)$.

Lemma 2 For each $\sigma \in \Sigma$, $t_i \in C$

$$\sigma(t_1,\ldots,t_n) \equiv \sigma_C(t_1,\ldots,t_n).$$

Proof The operators to consider are $\phi, a, x, hd, t\ell, cons, \leftarrow, \leftarrow_{hd}$ and $\leftarrow_{t\ell}$.

1. $\phi_C = \phi \equiv \phi$.

   $a_C = a \equiv a$.

2. Let $\sigma' = \leftarrow(y,c,\sigma)$.

   2.1 If $x = y$, c is a reference, then

   $x_C(\sigma') = c \equiv x(\sigma')$ by axiom 6.

   2.2 If $x = y$, c is a description, then

   $x_C(\sigma') = x(\sigma')$ and there is nothing to prove.

   2.3 If $x \neq y$

   $x_C(\sigma') = x(\sigma) \equiv x(\sigma')$ by axiom 6.

   Let $\sigma' = \leftarrow(\delta c_1, c_2, \sigma)$. Then

   $x_C(\sigma') = x(\sigma) \equiv x(\sigma')$ by axiom 5.

   Finally, $x_C(\phi) = \text{Error}_C \equiv x(\phi)$ by axiom 1.

3. Let $\delta \in \{hd, t\ell\}$.

   3.1 $\delta_C(cons(c_1,c_2))(\sigma) = c_i$ where $i = 1$ ff $\delta = hd, i=2$ if $\delta = t\ell$.

   $\equiv \delta(cons(c_1,c_2))$ by axiom 3 or 4.

3.2 $\delta_C(a)(\sigma) = \text{Error}_C$

$= \delta(a)(\sigma)$ by axiom 2.

3.3 Let $px(\sigma') \in C$, then we show by induction on σ that

$\delta_C(px(\sigma'))(\sigma) \equiv \delta(px(\sigma'))(\sigma)$

i) Let $\sigma = \phi$. Then

$\delta_C(px(\sigma'))(\sigma) = \text{Error} \equiv \delta(px(\sigma'))(\phi)$ by axiom 1.

ii) Suppose the result is true for σ. That is,

$\delta_C(px(\sigma'))(\sigma) \equiv \delta(px(\sigma'))(\sigma)$

for any $px(\sigma') \in C$.

a) Let $\sigma'' = \sigma' = \leftarrow(y,c,\ ),\ y \neq x$.

Then

$\delta_C(px(\sigma'))(\sigma'') = \delta_C(px(\sigma'))(\sigma)$

$\equiv \delta(px(\sigma'))(\sigma)$ by induction

$\equiv \delta(px(\sigma'))(\sigma'')$ by axiom 6.

b) Suppose $\sigma',\sigma'' \in C$, $\sigma' \neq \sigma''$, $\sigma'' = \leftarrow(y,c,\sigma)$.

Then if σ' is a subexpression of σ'',

$\delta_C(px(\sigma'))(\sigma'') = \delta_C(px(\sigma'))(\sigma)$ by definition.

$\equiv \delta(px(\sigma'))(\sigma)$ by induction.

$\equiv \delta(px(\sigma'))(\sigma'')$ by axiom 7.

If σ'' is a subexpression of σ', then

$\delta_C(px(\sigma'))(\sigma'') = \text{Error}$

$\equiv \delta(px(\sigma))(\sigma'')$ by axiom 7, since

$px(\sigma)$ is reduced.

If σ',σ'' are unrelated, then

$\delta_C(px(\sigma'))(\sigma'') = \delta(px(\sigma'))(\sigma'')$ by definition of δ_C.

c) Let $\sigma' = \sigma'' = \leftarrow(x,c,\sigma)$. Then

$\delta_C(px(\sigma'))(\sigma'') = \delta_C p_C c(\sigma)$

$= \delta_C \bar{p}_C \bar{c}(\sigma)$ for some $\bar{p}$ such that

$p = \bar{p}\,\bar{\bar{p}}$ since isref($\delta pc(\sigma)$).

$\equiv \delta\bar{p}\bar{c}(\sigma)$ by induction applied k times where $|\bar{p}| = k - 1$.

$\equiv \delta pc(\sigma)$ by applying axioms 3 and 4 k times.

d) Let $\sigma'' = \leftarrow(\delta c_1, c_2, \sigma)$.

If $c_1(\sigma) = px(\sigma')$, then

$\delta_C(px(\sigma'))(\sigma'') = c_2(\sigma)$

$\equiv \delta(px(\sigma'))(\sigma'')$ by axiom 9, since

$c_1(\sigma) = px(\sigma')$ implies $\simeq(c_1(\sigma), x(\sigma'))$.

If $c_1(\sigma) \neq px(\sigma')$, then since $c_1(\sigma), px(\sigma') \in C$,

$\simeq(c_1(\sigma), px(\sigma'))=$ false. Thus

$\delta(px(\sigma'))(\sigma'') \equiv \delta(px(\sigma'))(\sigma)$ by axiom 9

$= \delta_C(px(\sigma'))(\sigma'')$.

If $\bar{\delta} \neq \delta$

$\bar{\delta}_C(px(\sigma'))(\sigma'') = \bar{\delta}_C(px(\sigma'))(\sigma)$

$\equiv \bar{\delta}(px(\sigma'))(\sigma'')$ by axiom 8.

4. If $c_1, c_2 \in C$, then

$\text{cons}_C(c_1,c_2) = \text{cons}(c_1,c_2)$.

5. $\simeq_C(c_1,c_2) \equiv \simeq(c_1,c_2)$ follows directly from axiom 10.

6. $\leftarrow_C(x,c,\sigma) \quad = \leftarrow(x,c,\sigma)$

   $\leftarrow_C(\delta c_1,c_2,\sigma) = \leftarrow(\delta c_1,c_2,\sigma)$

7. If $p_1(p_2x(\sigma))(\vec{\sigma}) \in C$, then

   $\delta_C(p_1(p_2x(\sigma))(\vec{\sigma}))(\sigma_1) = \delta(p_1(p_2x(\sigma))(\vec{\sigma}))(\sigma_1)$

   and the result is immediate. □

Hence C does characterize the axioms, and in fact we can show

Theorem 3 $C \cong T_\Sigma/q$.

Proof Directly from lemmas 1 and 2 and Theorem 2.5. □

6 An Operational Model for Linked Lists

The Canonical Model C described in the previous section gives some aid in understanding the abstract model defined by the type axioms. C can also be used to establish the "correctness" of the axioms by finding an appropriate definitional model D and establishing

$$D \cong C.$$

However, in programming with the type linked list, we normally use a less general type. In this less general type a structure does not contain its "history" and hence two identical structures may arise by a different sequence of assignments. Assignments are restricted to be of the form $\leftarrow(x,c,\sigma)$ where σ is the only structure appearing in c, or of the form $\leftarrow_\delta(c_1,c_2,\sigma)$

where σ is the only structure appearing in c_1 and c_2. We define a model with these restrictions by presenting an algebra **M** whose semantics are the more usual semantics of linked lists. We then show that M satisfies the type axioms. (But note that M is not initial).

Definition 5 Let N be the set of natural numbers,

$$D = N + \text{Atom}$$

where + is a disjoint union. Let V be a set of objects called program variables. Then a simple structure is a pair of partial functions $<\ell,v>$

$$\ell\text{:}\quad N \to D * D \cup \{\text{Error}\}$$

$$v\text{:}\quad V \to N$$

such that there is some $m > 0$ for which

$$\ell(i) \in D \times D \qquad \text{if} \quad i \leq m$$

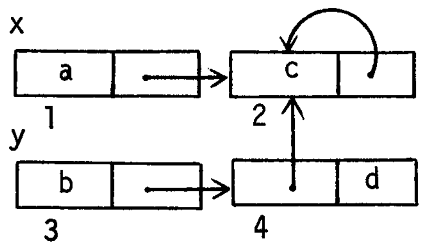
$$\ell(i) = \text{Error} \qquad \text{if} \quad i > m.$$

Let Card(ℓ) = m by definition. We also require that ff

$$\ell(i) = <j,k>$$

then $j,k \leq m$, and if $v(x)$ is defined, then $v(x) \leq m$. The elements of N are called addresses. v is said to associate an address with each variable.

Example 4 Consider the structure with diagram



This diagram represents the simple structure defined by

$\ell(1) = <a,2>$

$\ell(2) = <c,2>$

$\ell(3) = <b,4>$

$\ell(4) = <2,d>$

$v(x) = 1$

$v(y) = 3$ □

The operators Σ can be defined on simple structures.

Definition 6 Let π_i: $D \times D \to D$ $i = 1,2$ be defined by

$\pi_1(<d_1,d_2>) = d_1$

$\pi_2(<d_1,d_2>) = d_2$.

Let $s = <\ell,v>$ be a simple structure. Then we define

(i) $\phi_m = <\text{Error},v_\phi>$ where v_ϕ is the undefined function and Error is a constant function returning error.

(ii) $a_m = a$ for each $a \in M$.

(iii) $x_M(s) = v(x)$

(iv) $hd_M(i)(s) = \pi_1(\ell(i))$; $hd_M(a)(s) = \text{Error}$

(v) $t\ell_M(i)(s) = \pi_2(\ell(i))$; $t\ell_M(a)(s) = \text{Error}$

(vi) $cons_M(d_1,d_2) = <d_1,d_2>$

(vii) $$\approx_M(i,j) = \begin{cases} \underline{true} \text{ if } i = j \\ \underline{false} \text{ otherwise.} \end{cases}$$ □

Before defining the effect on structures, we need some auxiliary definitions.

<u>Definition 7</u> Let D = Atom + N. Then a <u>description in M</u> is

1. i) An element $d \in D$

   ii) A pair $<c_1,c_2>$ such that c_1 and c_2 are descriptions in M.

The set of descriptions will be denoted De. (Note that $D \subseteq De$).

2. We define a function

   no: De → N

   as follows:

   $no(d) = 0$ for $d \in D$

   $no(<c_1,c_2>) = 1 + no(c_1) + no(c_2)$

3. Let $<c_1,c_2>$ be a description, $m,i \in N$, $i > 0$. Then define

   allocate: $N \times De \times N \to D \times D$

   as follows

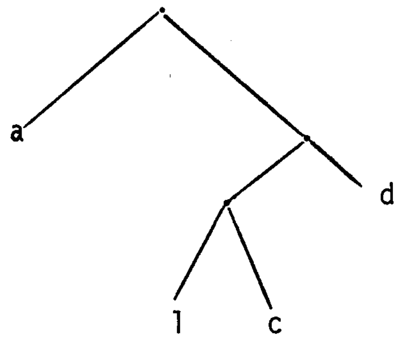
$$\begin{array}{lll}
\text{allocate}(m,\langle d_1,d_2\rangle,0) = \langle d_1,d_2\rangle & & d_1,d_2 \in D. \\
\text{allocate}(m,\langle c_1,d_2\rangle,0) = \langle m+1,d_2\rangle & & c_1 \notin D, d_2 \in D. \\
\text{allocate}(m,\langle d_1,c_2\rangle,0) = \langle d_1,m+1\rangle & & d_1 \in D, c_2 \notin D. \\
\text{allocate}(m,\langle c_1,c_2\rangle,0) = \langle m+1,m+no(c_1)+1\rangle & & c_1,c_2 \notin D.
\end{array}$$

$$\begin{array}{rl}
\text{allocate}(m,\langle c_1,c_2\rangle,i) = & \underline{\text{if}}\ i \le no(c_1)\ \underline{\text{then}}\ \text{allocate}(m+1,c_1,i-1) \\
& \underline{\text{else}}\ \text{allocate}(m+no(c_1)+1,c_2,i-(no(c_1)+1)).\square
\end{array}$$

The purpose of "allocate" is to assign new "storage" to each internal node in a tree. For example

$$c = (\text{cons}(a,\text{cons}(\text{cons}(1,c),d)))_M = \langle a,\langle\langle 1,c\rangle,d\rangle\rangle$$

which can be represented diagramatically as



Allocate assumes that the internal nodes of the tree are numbered in preorder (root, left son, right son) starting at m for the root. For example

$$\text{allocate}(m,c,0) = \langle a,m+1\rangle$$

$$\text{allocate}(m,c,1) = \text{allocate}(m+1,\langle\langle 1,c\rangle,d\rangle,0)$$

$$= \langle m+2,d\rangle$$

and $allocate(m,c,2) = allocate(m+1,\langle\langle 1,c\rangle,d\rangle,1)$

$$= allocate(m+2,\langle 1,c\rangle,0)$$

$$= \langle 1,c\rangle.$$

We establish the correctness of allocate by showing that the value of every symbolic traversal of an expression c is the same as the value obtained by successive "allocations" and projections.

For example

$$hd\ hd\ t\ell(cons(a,cons(cons(1,c),d)))(\sigma) = 1$$

and we require that if

$$a_0 = \pi_2(allocate(m,\langle a,\langle\langle 1,c\rangle,d\rangle\rangle,0))$$

$$a_1 = \pi_1(allocate(m,\langle a,\langle\langle 1,c\rangle,d\rangle\rangle,a_0-m))$$

and $\quad a_2 = \pi_1(allocate(m,\langle a,\langle\langle 1,c\rangle,d\rangle\rangle,a_1-m))$

then $\quad a_2 = 1.$

In general, π_2 corresponds to $t\ell$ (since $t\ell_M(i)(s) = \pi_2(\ell(i))$) and π_1 corresponds to hd (since $hd_M(i)(s) = \pi_1(\ell(i))$).

<u>Lemma 4</u> Let c be a description $c = \langle c_1,c_2\rangle$ and suppose that

$$a_0 = \pi_{i_0}(allocate(m,c,0))$$

$$a_j = \pi_{i_j}(allocate(m,c,a_{j-1}-m)) \quad 1 \le j \le n.$$

and $a_0,a_1,\ldots,a_n$ are defined, where $i_j \in \{1,2\}$. Then if $i_0 = 1$, then $a_{j-1}-m \le no(c_1)$ $\quad 1 \le j \le n$, and

if $i_0 = 2$, then $a_{j-1}-m > no(c_1)$.

<u>Proof</u> By induction on n. If n = 0 there is nothing to prove. If n = 1, then

$$a_0 = \pi_{i_0}(\text{allocate}(m,c,0))$$

$$a_1 = \pi_{i_1}(\text{allocate}(m,c,a_0-m))$$

Now if $c = \langle d_1,d_2\rangle$ $d_1,d_2 \in D$, then a_1 is not defined. If $c = \langle c,d\rangle$ $d \in D$, $c \notin D$ (or symmetrically, $\langle d,c\rangle$).then $a_0 = \pi_{i_0}(\langle m+1,d\rangle)$ (or $\pi_{i_0}(\langle d,m+1\rangle)$). If $i_0 = 2$ (= 1) then a_1 is not defined (in both cases). Hence suppose $i_0 = 1$ (1 = 2). Then

$$a_0 = m+1 \quad \text{(in both cases)}.$$

Thus $a_{1-1} = a_0 = m+1$, and $a_0 - m = 1 \le no(c_1)$ since $c_1 \notin D$ ($1 > no(d)$, since $d \in D$ and $no(d) = 0$).

Suppose that the result is true for $\langle a_0,a_1,\dots,a_n\rangle$, and suppose that

$$a_{n+1} = \pi_{i_{n+1}}(\text{allocate}(m,c,a_n-m)).$$

If $i_0 = 1$, then by induction $a_n-m \le no(c_1)$, so

$$a_{n+1} = \pi_{i_{n+1}}(\text{allocate}(m+1,c_1,a_n-(m+1))).$$

But note that

$$\max\{\pi_i(\text{allocate}(m',c,i)\} = m'+no(c)-1$$

for any i. Hence $a_{n+1} \le m+1+no(c_1)-1$

and so $a_{n+1}-m \leq no(c_1)$ as required.

If $t_0 = 2$, then by induction

$$a_n-m > no(c_1)$$

so $a_{n+1} = \pi_{i_{n+1}}(\text{allocate}(m+no(c_1)+1,c_2,a_n-(m+no(c_1)+1)))$.

But clearly $\min\{\pi_i(\text{allocate}(m,c,j))\} = m+1$, so

$$a_{n+1} \geq m+no(c_1)+1$$

so $a_{n+1} - m \geq no(c_1)+1$

and hence $a_{n+1} = m > no(c_1)$ as required. □

<u>Lemma 5</u> Let $p = \delta_n\delta_{n-1}\cdots\delta_0$ where $\delta_i \in \{hd,t\ell\}$, $0 \leq i \leq n$, and

let $i_j = \begin{cases} 1 & \text{if } \delta_j = hd \\ 2 & \text{if } \delta_j = t\ell \end{cases} \qquad 0 \leq j \leq n.$

Let c be a description in M with a subterm d in c such that

$$p_M(c)(s) = d$$

where $d \in D$, and let $\langle a_0,a_1,\ldots,a_n\rangle$ be a sequence, $a_i \in D$ for $0 \leq i \leq n$ where

$$a_0 = \pi_{i_0}(\text{allocate}(m,c,0))$$

$$a_1 = \pi_{i_1}(\text{allocate}(m,c,a_0-m))$$

$$\vdots$$

$$a_n = \pi_{i_n}(\text{allocate}(m,c,a_{n-1}-m)).$$

Then $a_n = d$.

<u>Proof</u> We establish this result by induction on the structure of c.

(i) If $c = \langle d_1, d_2 \rangle$, $d_1, d_2 \in D$, then the only possible strings p for which $p_M(c)$ is a subterm of c, $p_M(c) \in D$ are hd or $t\ell$. Then

$$\delta_M c(s) = d_i \quad \text{where } i = 1 \text{ if } \delta = \text{hd}, \; i = 2 \text{ otherwise.}$$

$$\begin{aligned} a_0 &= \pi_i(\text{allocate}(m, \langle d_1, d_2 \rangle, 0)) \\ &= \pi_i(\langle d_1, d_2 \rangle) \quad \text{by definition of allocate} \\ &= d_i . \end{aligned}$$

(ii) Suppose the result is true for descriptions c_1 and c_2.

a) Let $c = \langle c_1, d \rangle$ or $c = \langle d, c_1 \rangle$, $d \in D$, and let $p = \delta_n \ldots \delta_1 \delta_0$, where $\delta_0 = \text{hd}$ if $c = \langle c_1, d \rangle$, $\delta_0 = t\ell$ otherwise. (If this is not the case, then $p = \delta$, $\delta \in \{\text{hd}, t\ell\}$ and the result follows as in (i)). We assume that $c = \langle c_1, d \rangle$, $\delta_0 = \text{hd}$. The reverse case follows by symmetry.

Then

$$\begin{aligned} a_0 &= \pi_1(\text{allocate}(m, \langle c_1, d \rangle, 0)) \\ &= \pi_1(\langle m+1, d \rangle) = m+1 \\ a_1 &= \pi_{i_1}(\text{allocate}(m, \langle c_1, d \rangle, a_0 - m)) \\ &= \pi_{i_1}(\text{allocate}(m+1, c_1, 0)) \text{ by definition of} \\ &\quad \text{allocate and since } a_0 = m+1 \\ a_j &= \pi_{i_j}(\text{allocate}(m, \langle c_1, d \rangle, a_{j-1} - m)) \\ &= \pi_{i_j}(\text{allocate}(m+1, c_1, a_{j-1} - (m+1))) \text{ for } 1 < j \leq n \\ &\quad \text{by definition of allocate.} \end{aligned}$$

Now suppose that

$$(\delta_n \ldots \delta_0)_M(\langle c_1, d\rangle))(s) = d'.$$

Then $\quad (\delta_n \ldots \delta_1)_M(c_1)(s) \qquad = d'$ since $\delta_0 = hd$.

Letting $m' = m+1$, we must then have

$$a_n = d'$$

by induction on c_1.

b) Let $c = \langle c_1, c_2\rangle$ and let $p = \delta_n \ldots \delta_1 \delta_0$.

If $\delta_0 = hd$, then

$$(\delta_n \ldots \delta_1 \delta_0)_M\langle c_1, c_2\rangle(s) = (\delta_n \ldots \delta_1)_M(c_1)(s)$$
$$= d' \qquad \text{say.}$$

Also $a_0 = \pi_1(\text{allocate}(m,c,0)) = m+1$

and $a_j = \pi_{i_j}(\text{allocate}(m,c,a_{j-1}-m)) \qquad 1 \le j \le n.$

Now by lemma 4, since $i_0 = 1$, $a_{j-1}-m \le no(c_1)$, $1 \le j \le n$, so

$$a_j = \pi_{i_j}(\text{allocate}(m+1,c_1,a_{j-1}-(m+1))$$

and in particular

$$a_1 = \pi_{i_1}(\text{allocate } (m+1,c_1,m+1-(m+1))$$
$$= \pi_{i_1}(\text{allocate}(m+1,c_1,0)).$$

Hence, letting $m' = m+1$,

$$a_n = d' \qquad \text{by induction.}$$

If $\delta_0 = t\ell$, then

$$(\delta_n \ldots \delta_1 \delta_0)_M\langle c_1, c_2\rangle(s) = (\delta_n \ldots \delta_1)_M(c_2)(s)$$
$$= d' \qquad \text{say.}$$

Also $a_0 = \pi_2(\text{allocate}(m,c,0)) = m+\text{no}(c_1)+1$

$$a_1 = \pi_{i_1}(\text{allocate}(m,c,\text{no}(c_1)+1))$$
$$= \pi_{i_1}(\text{allocate}(m+\text{no}(c_1)+1,c_2,\text{no}(c_1)+1-(\text{no}(c_1)+1))$$
$$= \pi_{i_1}(\text{allocate}(m+\text{no}(c_1)+1,c_2,0).$$

and in general

$$a_j = \pi_{i_j}(\text{allocate}(m,c,a_{j-1}-m))$$
$$= \pi_{i_j}(\text{allocate}(m+\text{no}(c_1)+1,c_2,a_{j-1}-m-(\text{no}(c_1)+1))$$

since lemma 4 applies, $2 \le i_j \le n$.

Hence, letting $m' = m+\text{no}(c_1)+1$, we can apply induction, so

$a_n = d'$ as required □

We now define the assignment operators as follows:

Definition 8

1. Let $s = \langle \ell,v \rangle$ be a structure. Then

$$s' = \leftarrow_M(x,c,s) = \langle \ell',v' \rangle$$

is defined as follows: Let $m = \text{Card}(\ell)$

a) If c is a description, then

$$\ell'(i) = \begin{cases} \ell(i) & \text{if } i \le m. \\ \text{allocate}(m+1,c,i-m) & \text{if } i > m. \end{cases}$$

$$v'(y) = \begin{cases} m+1 & \text{if } x = y \\ v(y) & \text{otherwise.} \end{cases}$$

Note that $\text{Card}(\ell') = m+\text{no}(c)$.

b) If $c \in N$, then

$\ell'(i) = \ell(i)$.

$$v'(y) = \begin{cases} c & \text{if } x = y \\ v(y) & \text{otherwise.} \end{cases}$$

2. Let $s' = \leftarrow_M(\delta c_1, c_2, s) = <\ell', v'>$.

We define a function

$$\text{Update: } D \times D \times \{hd, t\ell\} \times D \to D \times D$$

as follows:

$$\text{Update}(<d_1,d_2>, hd, d_3) = <d_3, d_2>$$
$$\text{Update}(<d_1,d_2>, t\ell, d_3) = <d_1, d_3>.$$

Then $v' = v$

and

$$\ell'(i) = \begin{cases} \text{update}(\ell(i), \delta, c_2) & \text{if } i = c_1 \\ \ell(i) & \text{otherwise.} \end{cases}$$ □

As an algebra, M has carrier $<V, \text{Cell}_M, \text{Struct}_M>$ where V is a set of variables called program variables, $\text{Cell}_M = D \times D$ and Struct_M is the set of all structures as defined in definition 5. We can show that this simple model M satisfies the type axioms for linked lists. Before establishing the main theorem of this section we need the following auxiliary results.

Lemma 6 Suppose $s' = \leftarrow_M(x, c, s)$ where s is a description in M, $m' = \text{Card}(\ell')$, $m = \text{Card}(m)$. Then

(i) $\delta_M(i)(s') = \delta_M(i)(s) \quad \text{if} \quad i \le m$

(ii) For any $p \in \{hd, t\ell\}^*$

$p_M(i)(s') = p_M(i)(s) \quad \text{if} \quad i \le m$

(iii) If $m < i \le m'$, then there is no $q \in \{hd, t\ell\}^*$ $y \in V$, $y \neq x$ such that $q_M y_M(s') = i$, and there is a unique $p \in \{hd, t\ell\}^*$ such that $p_M x_M(s') = i$.

Proof

(i) Follows directly from the definition of δ_M.

(ii) Follows by succesive applications of (i).

(iii) Consider $q_M y_M(s')$, $y = x$.

$q_M y_M(s') = q_M(v'(y))(s')$

$= q_M(v(y))(s')$ by definition, since $x \neq y$

$= q_M(v(y))(s)$ by (ii) since $v(y) \le m$.

Finally, it is a property of binary trees that there is a unique traversal to a given node. So suppose

$\delta_n \ldots \delta_0$

is such a traversal. Then $q = \delta_n \ldots \delta_0$. □

Theorem 7 M satisfies the linked list axioms of definition 2.

Proof

1) We must show that

$\delta_M(q_M(s))(\phi) = \text{Error}$

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

But by definition $\delta_M(i)(\phi) = \text{Error}$ for any i, $\delta \in \{hd, t\ell\}$.

2) Follows since $\delta_M(a)(s) = \text{Error}$

3,4) $\quad cons_M(c_1, c_2) = \langle c_1, c_2 \rangle$

$\delta_M(\langle c_1, c_2\rangle)(s) = c_i$ where $i = 1$ if $\delta = hd$, $i = 2$ if $\delta = t\ell$

5) Follows since if

$$s' = \leftarrow_M(\delta c_1, c_2, s)$$

then $\quad x_M(s') = v'(x) = v(x) = x_M(s)$.

6) We must show that for any $q \in \{hd, t\ell\}^*$ if $s' = \leftarrow_M(x, c, s)$ and

$\sigma_M = s$, then $\quad q_M x_M(s') = \begin{cases} q_M c(s) & \text{if isref}(qc(\sigma)) \text{ and } y = x \\ q_M x_M(s') & \text{if } x \nequiv y. \end{cases}$

(i) If $x \nequiv y$, then

$$\begin{aligned} q_M x_M(s') &= q_M(v'(x))(s') && \text{(notation)} \\ &= q_M(v(x))(s') && \text{since } x \nequiv y \\ &= q_M(v(x))(s) && \text{by lemma 6} \\ &= q_M x_M(s). \end{aligned}$$

(ii) If $x = y$ and isref(qc) then

$$\begin{aligned} q_M x_M(s') &= q_M(v'(x))(s') \\ &= q_M(m+1)(s'). \end{aligned}$$

Now since isref(qc) there is a sequence $\bar{q} = \delta_n \ldots \delta_0$ such that

$$q = \overset{*}{q}\bar{q}$$

$(\delta_{n-1} \cdots \delta_0)_M(c)(s)$ is a non-atomic description

and $(\delta_n \ldots \delta_0)_M(c)(s) = d$ for $d \in N$, $d \in \text{range}(\ell)$.

Hence, by lemma 5

$$\begin{aligned} q_M(m+1)(s') &= (\bar{\bar{q}}\bar{q})_M(m+1)(s') \\ &= \bar{\bar{q}}_M(d)(s') \\ &= \bar{\bar{q}}_M(d)(s) \text{ since } d \le m\text{, as required.} \end{aligned}$$

7) We must show that

$$\begin{aligned} \delta_M(q_M x_M(s'))(s'') &= \delta_M(q_M x_M(s'))(s) \text{ if } s'' \text{ is derived from } \\ &\quad s' \text{ and } s'' = \leftarrow_M(y,c,s) \\ &= \text{Error if } s' \text{ is derived from } s'', \\ &\quad s' = \leftarrow_M(x,c,s)\text{, and } \sim\text{isref}(qc(s)). \end{aligned}$$

Suppose that s" is derived from s' by some sequence of assignments. Also, suppose that $q_M x_M(s') = j$. Now since s" is derived from s', $\ell_{s''}(j)$ is defined. Furthermore, since

$s'' = \leftarrow_M(y,i,s)$,

$\ell_{s''}(j) = \ell_s(j)$ follows by definition of $\leftarrow_M$.

If s' is derived from s", then $s' = \leftarrow_M(x,c,s)$ and $\sim\text{isref}(qc(s)) \Rightarrow q_M x_M(s') > \text{Card}(\ell_s)$. Hence $\ell_{s''}(j) = \text{Error}$.

8) Follows trivially.

9) We must show that

$$\begin{aligned} &\delta_M(p_{1M}(s'))(\leftarrow_M(\delta p_{2M}, p_{3M}, s)) \\ &\quad = \underline{\text{if}}\ p_{1M}(s') \simeq_M p_{2M}(s)\ \underline{\text{then}}\ p_{3M}(s) \\ &\quad\ \ \underline{\text{else}}\ \delta_M(p_{1M}(s'))(s). \end{aligned}$$

Now

$$p_{1M}(s') \simeq p_{2M}(s) \Rightarrow p_{1M}(s') = p_{2M}(s) = i \text{ (say)}.$$

Then $\delta_M(i)(s'') = \pi_j(\text{Update}(\ell(i),\delta,p_{3M}(s))$

$= p_{3M}(s)$ as required.

Finally, we need to show that

$p_{1M}(s) \neq p_{2M}(s) \Rightarrow p_{1M}(s) \not\simeq p_{2M}(s)$. But this follows by definition of $\simeq$. Hence the result follows.

10) We must show that

$${}_M(q_M x_M(s_1'),\ \bar{q}_M \bar{x}_M(s_2')) =$$

$$\underline{\text{if}}\ s_1' = s_2'\ \&\ \text{isref}(q_M c_1)\ \&\ {\sim}\text{isref}(q_M c_2)\ \underline{\text{then}}$$

$$qx = \bar{q}\bar{x}$$

$$\underline{\text{else}} \simeq_M(q_M x_M(s_1'),\ \bar{q}_M \bar{x}_M(s_2')).$$

Suppose that

$$s_1' = s_2' = s\ \&\ {\sim}\text{isref}(q_M c_1)\ \&\ {\sim}\text{isref}(q_M c_2).$$

Then

$$\simeq_M(q_M x_M(s),\ \bar{q}_M \bar{x}_M(s_2')) = q_M x_M(s) = \bar{q}_M \bar{x}_M(s)$$

by definition. If $qx = \bar{q}\bar{x}$, the result follows. We need to show that if $q_M x_M(s) = \bar{q}_M \bar{x}_M(s) = i$, then $qx = \bar{q}\bar{x}$. This follows since $i > m$, and hence there is a unique path in s to i, by lemma 6. □

Finally, we show that with respect to references, D and T_Σ/q are isomorphic.

Theorem 8

If $h(p_1x_1(\sigma_1)) = h(p_2x_2(\sigma_2))$ and $h(\sigma_1) = h(\sigma_2)$, then $p_1x_1 == p_2x_2$.

Proof: Let $h(p_1x_1(\sigma_1)) = (p_1x_1)_M s$; then $h(p_2x_2(\sigma_2)) = (p_2x_2)_M(s)$. So suppose that

$$(p_1x_1)_M(s) = (p_2x_2)_M(s) = i.$$

Now $p_1x_1(\sigma)$, $p_2x_2(\sigma) \in C_{Cell}$, hence σ is of the form $\leftarrow(x,c,\bar{\sigma})$, and $x_1 = x_2 = x$. Thus

$$(p_1x_1)_M(s) = (p_1)_M v_s(x)(s)$$

and $$(p_2x_2)_M(s) = (p_2)_M v_s(x)(s)$$

Furthermore, since $p_1x(\sigma)$ is canonical and $p_2x(\sigma)$ is canonical, $i > m_{\bar{s}}$ where $s = \leftarrow_M(x,c_M,\bar{s})$. Hence by lemma 6 (iii), $p_1 = p_2 = p$. □

7 Conclusions

The general type linked list has been defined as an abstract data type. In order to do this, it was necessary to treat the concept of referencing very carefully. Although the type axioms are fairly easy to read, they are complicated by the necessity to treat error conditions. However these axioms are straightforward to use in carrying out the kinds of reductions that occur in proving properties of programs, using the proof rule discussed in the first section of this chapter. In chapter 5 we prove the correctness of a simple linked list program, using the techniques presented in this chapter. We illustrate how the language of the type can be used to express properties of linked list structures. The type presented here is more general than the type usually implemented as linked list, and one might expect that this generality complicates the type axioms and the use of the axioms. We feel that the reverse is the case. That is, both the axioms and the use of them is simplified by using this type rather than a more restricted type. It is interesting to ask whether the specification can be found for the more restricted type. That is, given Σ as above, is there a set of equations ε which generate a congruence q such that

$$M \cong T_{\Sigma}/q \ ?$$

Although it is possible that such a specification may exist, we believe the specification in this chapter to be much more useful, at least from the point of view of verification. In the next chapter, the problem of sharing in a more general setting is studied.

Chapter 4

Sharing and Circularity With Continuous Data Types

1. Data Structures with Sharing

Central to this thesis is the idea that data types are algebras of some form. In this chapter we consider data types to be continuous algebras and extend the results needed to characterize data types to continuous algebras. A data structure is viewed as being some element of a data type. Suppose for example that A is an algebra (data type) with some element a which will be named x. Suppose also that A is a continuous Σ-algebra. Then, since CT_Σ is initial in the class of continuous Σ-algebras, we can characterize the value of $a \in A$ with any $t \in CT_\Sigma$ such that $h(t) = a$. We write

$$x = t$$

to denote the fact that the value of x in A is $h(t)$. Suppose now that $X_n = \{x_1,\ldots,x_n\}$ is a set of n distinct elements called variables. In some computation, at some particular time, each variable will have associated with it a particular value. (This association is often called the <u>state</u> of the computation, although the state will in general include additional information such as the association of variables of other types and some kind of program counter.)

We indicate the correct state of the variables of some particular type by a set of equations of the form

$$x_1 = t_1$$
$$x_2 = t_2$$
$$\vdots$$
$$x_n = t_n$$

where $t_i \in CT_\Sigma$, $1 \leq i \leq n$. This association can also be denoted by a function

$$e:X_n \to CT_\Sigma$$

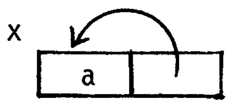
such that

$$e\,(x_i) = t_i\ .$$

We call such a function e a <u>structure</u>. A structure has <u>sharing</u> if the value associated with same variable x_i is itself dependent on the value of some other variable, say x_j. Then if some component of x_j is changed, we would like this change to be reflected in the value of x_i . To achieve this, consider a structure to be a function

$$e:X_n \to CT_\Sigma(X_n)$$

so that $e(x_i) = t_i$, and t_i may contain variables. For example, consider the simple circular list structure



This can be represented by the simple equation

$$x = \mathrm{cons}(a,x).$$

However, there is no unique homomorphism

$$h\colon\ CT_\Sigma(X_n) \to A$$

since the value of h is affected by assignment to X_n. It is known that any set of equations

$$e\colon\ X_n \to CT_\Sigma(X_n)$$

has a "solution" in CT_Σ. This solution is an n-tuple, $\bar{t} = \langle \bar{t}_1, \ldots, \bar{t}_n \rangle$ with the property (among others) that if $e_i(\bar{t})$ denotes the element of CT_Σ obtained from $e(x_i)$ by replacing each occurrence of x_j in $e(x_i)$ by $\bar{t}_j$, then $e_i(\bar{t}) = \bar{t}_i$. Now since $\bar{t}_i \in CT_\Sigma$, there is a corresponding tuple $a = \langle a_1, \ldots, a_n \rangle$, $a_i \in A$ such that $h(\bar{t}_i) = a_i$, where h is the unique homomorphism

$$h\colon\ CT_\Sigma \to A\ .$$

We can thus characterize the "value" of any $t \in CT_\Sigma(X_n)$ in A as the image of the homomorphism extending the assignment

$$x_i \mapsto a_i\ .$$

Hence, $t_A = \bar{a}(t)$ where $\bar{a}$ is the unique homomorphism

$$\bar{a}: \; CT_\Sigma(X_n) \to A$$

extending the assignment. Then the value of a structure e in A is $<\bar{a}(\bar{t}_1), \bar{a}(\bar{t}_2), \ldots, \bar{a}(\bar{t}_n)>$ which can be shown to be equal to $<\bar{a}(x_1), \bar{a}(x_2), \ldots, \bar{a}(x_n)>$. Thus a structure e completely determines the value in A associated with each variable $x_i \in X_n$. (This is called the solution of e in A, often denoted $|e_A|$.)

Suppose for example that the type in question is N, the type integers, and that we want two variables x_1 and x_2 to share their value. For example, let e be

$$x_1 = x_2$$
$$x_2 = 9 \times 3 .$$

Then the solution to this set of equations in N is

$$<27,27>$$

so that, for example, if

$$t = x_1 + x_2 - 20$$

then $\quad h_A(t) = 27 +_N 27 -_N 20 = 34$.

Note that the solution of e in CT_Σ is

$$<9 \times 3, \; 9 \times 3> .$$

(It is possible to have a more "elaborate" sharing between x_1 and x_2. For example, the structure

$$x_1 = x_2 + 3$$

$$x_2 = 9 \times 3$$

indicates that the value of x_1 is 3 larger than x_2. If the value of x_2 is changed, then the value of x_1 is also changed. This type of "sharing" is not usually used for primitive data types, although it is used for more complex types.)

The important point to note about the above example is that the solution of e has "lost" the shared information. That is, it is not possible to determine from the solution of e in N that x_1 is necessarily related to x_2. This is the reason why we regard e as the data structure, rather than an association

$$a: X_n \to A$$

which is more usual. (For example, in the state vector approach of Scott and Strachey [38], each variable has an associated value in the type of values V.) If there is no sharing, then e and a can be used interchangably: e contains no information that cannot be derived from a.

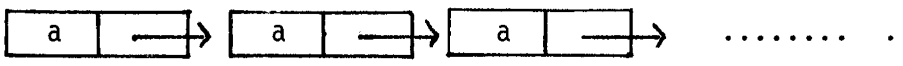
Having settled on the notion of a data structure with sharing, a number of questions must still be answered: for example, how is the type (algebra) A to be characterized? In chapter 2, it was shown that types could be characterized as quotients on T_Σ. However, in order for the function e to have solutions in general, (even if

$$e\colon\ X_n \to T_\Sigma(X_n), \text{ that is the right hand sides are finite)}$$

it is necessary to restrict attention to the class of continuous algebras rather than considering the class of all Σ-algebras. For example, consider the equation

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

which was viewed above as the structure of a circular list. This equation has no solution in T_Σ, and so the mechanism of computing the value of x in some algebra A in terms of extended homomorphisms cannot be applied to T_Σ. However, there is a solution in CT_Σ: this solution can be viewed intuitively as evaluating to the "infinite" list



It has been shown (for example, Reynolds [35]) that the data type lists can be regarded as a continuous algebra. (In fact as a complete partial order with continuous operators.) Is it possible to characterize such continuous data types as quotients on CT_Σ, the initial Σ-algebra in the class of all continuous Σ-algebras? A final question that must be treated is the question of updates. How can the semantics of updates be defined in terms of these continuous data types? We have suggested that the solution (or value) of a complex data structure does not contain sufficient information to define updating correctly. The nature of the sharing must somehow be retained. Is it possible for this to be done?

It has been suggested (Ashcroft [5]), that the equations e can be regarded as the data structure and that updating can be defined in terms of transformations to the equations. This is the basic idea that has been used for the treatment of shared types described in this chapter.

In the next section, we generalize some of the results of quotients and data types to continuous algebras. In particular we show that if ε is a set of Σ-equations generating a congruence q, and if there exists a function that selects "normal forms" of classes in q, then CT_Σ/q is initial in the class of all continuous Σ-algebras which satisfy ε. We then present the axioms (E) for a type <u>List</u> of list structures which may include infinite lists (obtained from circularities). We show that there is a normal form function for this type, and hence CT_Σ/q_E is initial. We then show that any particular structure e imposes an algebraic structure on CT_Σ/q_E, and that this algebraic structure may be characterized by a congruence q_e on CT_Σ/q_E. We prove that the quotient of CT_Σ/q_E by q_e is initial in the class of all continuous Σ-algebras satisfying E and e together, and in fact characterize equivalence in q_e in terms of equivalence in q_E . We then regard the initial algebra in the class of all continuous Σ-algebras satisfying E and e together as a data structure, and define a data type LIST whose objects are these data structures (with sharing). The operators of LIST include a generalized update operator R which updates data structures. (R can be viewed as a generalization of CSET, REPLACA and REPLACD). Finally we show that R is continuous with respect to the carrier of LIST, and hence that LIST is continuous.

2. An Initial Algebra for Continuous Data Types

2.1 Introduction

It is very tempting to try to generalize the work in non-continuous data types and establish the same results for continuous algebras. Thus one might hope that if ε is a set of equations generating a congruence q, then CT_Σ/q is initial in the appropriate class of Σ-algebras satisfying ε. Unfortunately this result is not always true. The reason for this is that CT_Σ/q is not always complete.

For example, consider an algebra with a single sort S and operators

$$i: S \to S \qquad \text{for each } i \in \omega$$

$$a, \perp: \to S$$

and suppose that q is the least ω-continuous congruence defined by the equations

$$i.a = i + 1 . \perp \qquad \text{for each } i \in \omega.$$

Now suppose that a partial order $\sqsubseteq$ on CT_Σ/q is defined as follows:

$$[t_1] \sqsubseteq [t_2] \iff \perp \in [t_1] \text{ or } u.i.a \in [t_1] \text{ and } u.j.a \in [t_2], \; u \in \omega^* \text{ and } i \leq j .$$

$\sqsubseteq$ is clearly a partial order. Furthermore, it is consistent with the ordering on CT_Σ since all chains in CT_Σ/q are of the form

$$[u.i.a] \sqsubseteq [u.j.a]$$

where $j \geq i$ and if $j > i$, then there always exists a finite chain

$$[u.i.a] = [u.i{+}1.\perp] \sqsubseteq [u.i{+}1.a] = [u.i{+}2.\perp] \sqsubseteq \ldots \\ \ldots [u.j{-}1.\perp] = [u.j.a].$$

But note that the chain $<[u.i.a]>_{i\in\omega}$ has a least upper bound in CT_Σ/q if and only if $<i>_{i\in\omega}$ has a least upper bound in ω. Since no such upper bound exists, CT_Σ/q cannot be complete.

However, most data types are probably well behaved, and so we show a weaker result which should be useful for a large class of data types.

2.2 Normal Forms

For practical reasons, it is often useful to try to characterize a class of values which are equivalent in some equivalence relation by a single representative of the class. We call such a representative a normal form of the class. This practical consideration in fact leads to a sufficient condition for guaranteeing the initiality of CT_Σ/q .

Definition 1   Suppose there exists a function

$$nf: \ CT_\Sigma \to CT_\Sigma$$

such that for $t, t_1, t_2 \in CT_\Sigma$, and a congruence q,

1. $[t_1] = [t_2] \Rightarrow nf(t_1) = nf(t_2)$;
2. $[nf(t)] = [t]$;
3. nf is Δ-continuous (in the usual ordering on CT_Σ).

$[t] = \Theta(t)$ where Θ is the natural homomorphism induced by the congruence q. nf is called a normalizer function for CT_Σ/q.
We define, for $[t_1], [t_2] \in CT_\Sigma/q$

$$[t_1] \sqsubseteq_q [t_2] \iff nf(t_1) \leq_{CT} nf(t_2)$$

where $\leq_{CT}$ is the partial order on CT_Σ .
(Where no ambiguity can arise, the subscript CT of $\leq_{CT}$ and the

subscript q of $\sqsubseteq_q$ will be dropped). $\square$

Note that nf need not be unique, and so the order relation $\sqsubseteq_q$ also depends on nf.

Berry and Courcelle [7] have independently described a similar mapping c called a c-projection. The intent of a c-projection is also to find a normal form of the free algebra in question, but it is assumed by them that an order relation is defined in the target algebra. However, the normalizer here is used to define the order relation on CT_Σ/q. In ADJ [1], a "Canonical Term Algebra" is defined. It is an algebra whose carrier corresponds more or less to the sets of normal forms defined by the range of nf. In fact it is shown in ADJ [1] that at least for T_Σ, every data type described by a set of equational axioms has a canonical term algebra.

Note that in this and subsequent sections, we are assuming only a single sort for CT_Σ. This is simply for notational convenience: the results all still hold for the many sorted case. Thus for example we should say that nf is a family of functions $\langle nf_s \rangle_{s \in S}$, one for each sort $s \in S$.

Lemma 1 i) $nf(nf(t)) = nf(t)$. That is nf is idempotent.

ii) $nf(t_1) = nf(t_2) \Rightarrow [t_1] = [t_2]$.

That is, two normal forms are equal only if the corresponding terms are equivalent.

Proof i) From property 2 we get $[nf(t)] = [t]$

so $nf(nf(t)) = nf(t)$ from property 1.

ii) Suppose $nf(t_1) = nf(t_2)$. Then

$[t_1] = [nf(t_1)] = [nf(t_2)] = [t_2]$ □

Lemma 2 $\sqsubseteq_q$ is a partial order on CT_Σ/q.

Proof Since $\leq$ is a partial order on CT_Σ. □

The subscript q in $\sqsubseteq_q$ will be dropped when there is no possible ambiguity.

Lemma 3 Let $<t_i>_{i \in I}$ be a set directed in CT_Σ. Then $<[t_i]>$ is directed in CT_Σ/q and it has a least upper bound denoted $\sqcup_i [t_i]$ such that $\sqcup_i [t_i] = [\sqcup_i t_i]$.

Proof Let $t = \sqcup_i t_i$, which exists since CT_Σ is Δ-complete. By monotonicity of nf and definition of $\sqsubseteq$, $<[t_i]>$ is directed. Now for all $i \in I$, $t_i \leq t$ since t is the least upper bound of $<t_i>_{i \in I}$.

Hence $\forall i \; nf(t_i) \leq nf(t)$

so $\forall i \; [t_i] \sqsubseteq [t]$ by definition of $\sqsubseteq$.

So $[t]$ is an upper bound of $<[t_i]>_{i \in I}$. Now suppose that for some

$\bar{t}$, for all $i \in I$, $[t_i] \sqsubseteq [\bar{t}]$. Then for all $i \in I$, $nf(t_i) \leq nf(\bar{t})$. But since nf is Δ-continuous and $\langle t_i \rangle_{i \in I}$ is directed, $\langle nf(t_i) \rangle_{i \in I}$ is directed and

$$\sqcup_i\, nf(t_i) \leq nf(\bar{t})$$

$\therefore$ $nf(\sqcup_i t_i) \leq nf(\bar{t})$ by continuity of nf

$\therefore$ $[\sqcup_i t_i] \sqsubseteq [\bar{t}]$ by definition of $\sqsubseteq$.

Hence $[\sqcup_i t_i]$ is the least upper bound of $\langle [t_i] \rangle_{i \in I}$. That is

$$\sqcup_i [t_i] = [\sqcup_i t_i]. \qquad \square$$

<u>Lemma 4</u> If B is a Δ-continuous Σ-algebra satisfying ε, q is the congruence generated by ε, $t_1, t_2 \in CT_\Sigma$ and $(t_1, t_2) \in q$ and

$$h_B\colon\ CT_\Sigma \to B$$

is the unique homomorphism gauranteed to exist by the initiality of CT_Σ, then $h_B(t_1) = h_B(t_2)$.

<u>Proof</u> Let $Ker(h_B) = \{\langle t_1, t_2 \rangle \in CT_\Sigma \times CT_\Sigma | h_B(t_1) = h_B(t_2)\}$ be the kernel of h_B. Clearly this is an equivalence relation on CT_Σ and in fact it is a congruence:

For $\sigma \in \Sigma_n$, $t_1, \ldots, t_n \in CT_\Sigma$, $t_1', \ldots, t_n' \in CT_\Sigma$

such that $(t_i, t'_i) \in Ker(h_B)$, $1 \le i \le n$, we have

$h_B(\sigma(t_1, \ldots, t_n)) = \sigma(h_B(t_1), \ldots, h_B(t_n))$ since h_B is a homorphism

$= \sigma(h_B(t'_1), \ldots, h_B(t'_n))$ by definition of the kernel

$= h_B(\sigma(t'_1, \ldots, t'_n))$ and hence

$(\sigma(t_1, \ldots, t_n), \sigma(t'_1, \ldots, t'_n)) \in Ker(h_B)$.

Furthermore, $Ker(h_B)$ is continuous since if $(t_i, t'_i) \in Ker(h_B)$, $i \in I$, then

$h_B(\sqcup_i t_i) = \sqcup_i h_B(t_i)$ by continuity of h_B

$= \sqcup_i h_B(t'_i)$ definition of $Ker(h_B)$

$= h_B(\sqcup_i t'_i)$ once again by continuity.

Thus $(\sqcup_i t_i, \sqcup_i t'_i) \in Ker(h_B)$. Finally, since B satisfies ε, $\varepsilon(B) \subseteq Ker(h_B)$, where $\varepsilon(B)$ is the relation on B generated by the set of equations ε. This follows because for each assignment

$\Theta\colon X \to B$

and each $\langle L,R \rangle \in \varepsilon$, $\bar{\Theta}(L) = \bar{\Theta}(R)$. Now $\bar{\Theta} = h_B$ by uniqueness of h_B. Hence $h_B(L) = h_B(R)$.

But q is the least continuous Σ-congruence containing $\varepsilon(B)$. Hence $q \subseteq Ker(h_B)$, and so $(t_1, t_2) \in Ker(h_B)$ and hence $h_B(t_1) = h_B(t_2)$ as required. □

2.3 Initiality

Much of the power of considering abstract data types as many-sorted algebras centres around the property of isomorphism. Different implementations of the same data type can be considered members of a class of isomorphic algebras. In order to characterize this class precisely, the concept of initial algebra is used. The initial algebra in a class of algebras contains in some sense the least amount of information needed to specify a member of the class. Thus we would like to say that a particular abstract data type is the initial algebra in a class of algebras satisfying the specifications. Initiality ensure that the operators do no more than required by the specification. ADJ [1] shows that T_Σ/q is initial in the class of algebras satisfying the equations which generate the congruence q. It is natural to ask whether or not CT_Σ/q is initial, and we show that if the normalizer nf exists, then indeed CT_Σ/q is initial.

Definition 2 The class of all Σ-algebras that are ω-continuous with Δ-continuous Σ-homomorphisms and that satisfy ε will be denoted

$$\underline{\underline{\Delta\mathrm{Alg}}}_{\Sigma,\varepsilon} \qquad \Box$$

Theorem 5 If a normalizer nf exists for q, then CT_Σ/q is initial in $\underline{\underline{\Delta\mathrm{Alg}}}_{\Sigma,\varepsilon}$

Proof We must find a unique

$$h_B : CT_\Sigma/q \to B$$

for any $B \in \underline{\underline{\Delta Alg}}_{\Sigma,\varepsilon}$. By Theorem 2.6, $h_1 : CT_\Sigma \to B$ exists, and is unique. Now define

$$h_B([t]) = h_1(nf(t)).$$

i) h_B is a Σ-homomorphism. We must show that

$$h_B([\sigma(t_1,\ldots,t_n)]) = \sigma(h_B([t_1]),\ldots,h_B([t_n]))$$

for any $\sigma \in \Sigma_n$.

$h_B([\sigma(t_1,\ldots,t_n)]) = h_1(nf(\sigma(t_1,\ldots,t_n)))$ by definition of h_B.

$= h_1(\sigma(t_1,\ldots,t_n))$ by lemma 4 and property 2 of nf

$= \sigma(h_1(t_1),\ldots,h_1(t_n))$ since h_1 is a homomorphism

$= \sigma(h_1(nf(t_1)),\ldots,h_1(nf(t_n)))$ again by lemma 4 and property 2 of nf.

$= \sigma(h_B([t_1]),\ldots,h_B([t_n]))$ by definition of h_B.

ii) h_B is unique. Suppose there is a $g_B : CT_\Sigma/q \to B$ such that g_B is a homomorphism. Now consider the following diagram:

$$\begin{tikzcd}
CT_\Sigma \arrow[rr, "\Theta"] \arrow[d, "h_1"'] & & CT_\Sigma/q \arrow[dll, shift right, "h_B"'] \arrow[dll, shift left, "g_B"] \\
B & &
\end{tikzcd}$$

where Θ is the natural homomorphism induced by q. If g_B exists, then we must have $\Theta \circ h_B = \Theta \circ g_B = h_1$

since $\Theta \circ h_B$ and $\Theta \circ g_B$ are both homomorphisms into B. But Θ is onto, hence $h_B = g_B$.

iii) CT_Σ/q is Δ-complete. Let $<[t_i]>_{i\in I}$ be directed in CT_Σ/q. Then by definition of $\sqsubseteq$, $<nf(t)>_{i\in I}$ is directed in CT_Σ. Applying lemma 3 and since $[t] = [nf(t)]$ for any $t \in CT_\Sigma$, we get

$$\begin{aligned} \sqcup_i[t_i] = \sqcup_i[nf(t_i)] &= [\sqcup_i nf(t_i)] \\ &= [t] \end{aligned}$$

where $t = \sqcup_i nf(t_i)$ exists since CT_Σ is Δ-complete.

iv) CT_Σ/q is Δ-continuous. By (iii) CT_Σ/q is complete. $[\perp]$ is the minimum element of CT_Σ/q since nf is continuous and hence $nf(\perp) = \perp$. Now we must show that for each $\sigma \in \Sigma_n$, and each $1 \le j \le n$,

$$\sigma([t_1],\dots,\sqcup_i[t^i_j],\dots,[t_n]) = \sqcup_i\sigma([t_1],\dots,[t^i_j],\dots,[t_n]).$$

$\sigma([t_1],\ldots,\sqcup_i[t_j^i],\ldots,[t_n])$

$= \sigma([t_1],\ldots,[\sqcup_i t_j^i],\ldots,[t_n])$ by lemma 3

$= [\sqcup_i(\sigma(t_1,\ldots,t_j^i,\ldots,t_n))]$ by definition and continuity of σ

$= \sqcup_i[\sigma(t_1,\ldots,t_j^i,\ldots,t_n)]$ by lemma 3

$= \sqcup_i(\sigma([t_1],\ldots,[t_j^i],\ldots,[t_n]))$ by definition of σ

v) h_B is Δ-continuous. We must show that if $[t]$ is the least upper bound of a directed set $<[t_i]>_{i\in I}$ in CT_Σ/q, then

$$h_B([t]) = \sqcup_i h_B([t_i]).$$

By lemma 3 and part iii above we know that if $<[t_i]>_i$ is directed, and if $t = \sqcup_i nf(t_i)$, then

$$\sqcup_i[t_i] = \sqcup_i[nf(t_i)] = [t] = [\sqcup_i t_i] = [\sqcup_i nf(t_i)] \quad .$$

Now $h_B([t]) = h_1(nf(t))$ by definition of h_B

$= h_1(nf(\sqcup_i nf(t_i)))$ by definition of t

$= h_1(\sqcup_i nf(t_i))$ since nf is continuous and idempotent

$= \sqcup_i h_1(nf(t_i))$ since h_1 is continuous

$= \sqcup_i h_B([t_i])$ by definition of h_B as required.

□

The particular normalizer chosen will not affect the ordering on CT_Σ/q, because any two initial algebras must be isomorphic and have the same structure.

In practice, an algebra of normal forms is useful for establishing properties about an abstract data type and we make the following definition.

<u>Definition 3</u> A Δ-continuous Σ-algebra L_Σ is called a <u>normal term algebra</u> for q if

i) The carrier of L_Σ is a subset of the carrier of CT_Σ,

and ii) $L_\Sigma \cong CT_\Sigma/q$. □

If a normalizer exists, then it is possible to construct a normal term algebra.

<u>Theorem 6</u> Let

$$nf : CT_\Sigma \to CT_\Sigma$$

be a normalizer, and define a Σ-algebra L_Σ as follows

i) The carrier is L,

$$L = \{nf(t) \mid t \in CT_\Sigma\}$$

ii) For each $\sigma \in \Sigma$

$$\sigma_L(nf(t_1),\ldots,nf(t_n)) = nf(\sigma(t_1,\ldots,t_n)).$$

Then L_Σ is a normal term algebra.

<u>Proof</u> Let $g : L_\Sigma \to CT_\Sigma/q$ be defined as the restriction of the natural homomorphism Θ to L_Σ.

i) g is a homomorphism, since Θ is.

ii) g is surjective since if $[t] \in CT_\Sigma/q$, then

$[t] = [nf(t)]$ since nf is a normalizer

and so $g(nf(t)) = [t]$ by definition of g.

iii) g is injective, since if

$g(nf(t_1)) = g(nf(t_2))$ then

$[nf(t_1)] = [nf(t_2)]$ by definition of g, and so

$nf(t_1) = nf(t_2)$ since nf is a normalizer. □

We now demonstrate the converse of this theorem.

Theorem 7 If a normal term algebra L_Σ exists, then there is a normalizer function for q.

Proof By initiality of CT_Σ, $h:CT_\Sigma \to L_\Sigma$ exists. Let $nf(t) = h(t)$. We must show that

i) $[t_1] = [t_2] \Rightarrow nf(t_1) = nf(t_2)$

This follows by lemma 4 since L_Σ satisfies ε.

ii) $[nf(t)] = [t]$.

That is, we must show that $[h(t)] = [t]$.

Since $L_\Sigma \subseteq CT_\Sigma$, $h \circ h = h$. Also, since

$L_\Sigma \cong CT_\Sigma/q$, $Ker(h) = q$. Now since

$h \circ h(t) = h(t)$, $(h(t),t) \in Ker(h)$

so $(h(t),t) \in q$

and hence $[h(t)] = [t]$ as required.

iii) Continuity of nf is immediate since h is continuous.

□

3. The Data Type List

We would now like to apply these ideas to characterize an important class of structures: the so-called arbitrary list structures. We present the usual axioms of the type and show that there is an initial algebra in the class of algebras satisfying these axioms.

Definition 4 The class of algebras to be considered has a single sort called List with operators

$a: \to \text{List}$	for each $a \in \text{Atom}$ where Atom is an arbitrary set.
$x: \to \text{List}$	for each $x \in X$, a countable set of elements called <u>variables</u>.
$\text{cons}: \text{List} \times \text{List} \to \text{List}$	
$\text{hd}, t\ell: \text{List} \to \text{List}$	
$\bot: \to \text{List}$	a constant called <u>undefined</u>.

The axioms ε are: (Assume ℓ_1, ℓ_2 are variables of sort List and $a \in \text{Atom}$.)

1) $\text{hd}(\text{cons}(\ell_1,\ell_2)) = \ell_1$

2) $t\ell(\text{cons}(\ell_1,\ell_2)) = \ell_2$

3) $\text{hd}(a) = \bot$ for each $a \in \text{Atom}$

4) $t\ell(a) = \bot$ for each $a \in \text{Atom}$

5) $\text{hd}(\bot) = \bot$

6) $t\ell(\bot) = \bot$ □

Definition 5 A term $t \in CT_\Sigma$ is said to be reduced if it is of one of the forms

i)	$\perp$	
ii)	a	for any $a \in$ Atom
iii)	$cons(\ell_1,\ell_2)$	and ℓ_1,ℓ_2 are reduced
iv)	px	where $p \in \{hd, t\ell\}^*$
or v)	$t = \sqcup_i t_i$	for a directed set $<t_i>$ in CT_Σ, where each t_i, $i \in I$, is reduced. □

Definition 6 L_Σ for lists is defined as follows:

i) The carrier of L_Σ is the set of reduced terms of CT_Σ.

ii) The operations are defined as follows:

a) $a_L = a$ for each $a \in$ Atom

b) $x_L = x$ for each $x \in X$

c) $cons_L(t_1,t_2) = cons(t_1,t_2)$

d) $hd_L(cons(t_1,t_2)) = t_1$

$hd_L(px) = hd px$ for any $p \in \{hd, t\ell\}^*$

$hd_L(a) = \perp$

$hd_L(\perp) = \perp$

e) $t\ell_L(cons(t_1,t_2)) = t_2$

$t\ell_L(px) = t\ell px$ for any $p \in \{hd, t\ell\}^*$

$t\ell_L(a) = \perp$

$t\ell_L(\perp) = \perp$

f) $\perp_L = \perp$ □

Note the following points about L_Σ:

i) The carrier of L_Σ is a subset of the carrier of CT_Σ;

ii) The order relation $\leq$ on L_Σ is defined to be the restriction of $\leq_{CT}$ to L_Σ;

iii) L_Σ satisfies the axioms of definiton 4.

iv) By condition (v) of definition 5, L_Σ is Δ-complete, and by continuity of CT_Σ, L_Σ is easily seen to be Δ-continuous.

<u>Definition 7</u> Define a function

$$r:T_\Sigma \to T_\Sigma$$

as follows

$$r(px) = px$$
$$r(p\ \delta_i\ \mathrm{cons}(t_1,t_2)) = r(pt_i) \quad \text{where} \quad \delta_1 = \mathrm{hd},\quad \delta_2 = t\ell\ .$$
$$r(\mathrm{cons}(t_1,t_2)) = \mathrm{cons}(r(t_1),\ r(t_2))$$
$$r(p\bot] = \bot$$
$$r(a) = a$$
$$r(pa) = \bot$$

where $p \in \{\mathrm{hd},\ t\ell\}^*$. r is said to <u>reduce</u> any term $t \in T_\Sigma$.

<u>Lemma 8</u> If $t \in T_\Sigma$, with r defined as above, then

i) $r(t)$ is reduced and $(t,r(t)) \in q$

and ii) r is monotonic - that is, for any $t' \in T_\Sigma$ such that $t' \leq t$, $r(t') \leq r(t)$.

<u>Proof</u> By induction on the depth of t, which will be denoted $||t||$.

a) Let $t \in \{a,x,\perp\}$. Then $r(t) = t$ so clearly i) and ii) are true.

b) Suppose the result is true for any tree t of depth n or less. That is, if $||t|| \leq n$, then

i) $r(t)$ is reduced and $(t,r(t)) \in q$

and ii) If $t_1 \leq t_2$, $||t_2|| \leq n$, then $r(t_1) \leq r(t_2)$.

Now consider

1) δt $\delta \in \{hd, t\ell\}$

Since t is reduced, it is of the form px or $cons(t_1,t_2)$ where t_1,t_2 are reduced, or it is x or a or $\perp$. The last 3 cases have been shown. If $t = px$, then $r(\delta t) = \delta t$ and i) and ii) follow. If $t = cons(t_1,t_2)$, then $r(\delta t) = t_i$, but by axioms 1 or 2 $\delta t = t_i$ so i) follows, where $i = 1$ if $\delta = hd$, $i = 2$ if $\delta = t\ell$.

Suppose that $t' \leq \delta t$. Then $t' = \perp$ or $t' = \delta t''$ where $t'' \leq t$.

Hence $t'' = \perp$ or $t'' = cons(t_1'',t_2'')$ with

$t_1'' \leq t_1$, $t_2'' \leq t_2$.

Now

$r(t') = \perp$ in which case ii) follows;

or $r(t') = r(t_i'')$.

Also, $r(t) = r(t_i)$ and hence, by induction

$r(t_i'') \leq r(t_i)$.

Hence $r(t') \leq r(t)$ as required.

2) Consider $cons(t_1,t_2)$ where $||cons(t_1,t_2)|| = n + 1$.

$r(cons(t_1,t_2)) = cons(r(t_1), r(t_2))$ so i) follows by induction.

Suppose

$t' \leq cons(t_1,t_2)$

for some $t' \in T_\Sigma$.

If $t' = \perp$, there is nothing to prove, so let

$t' = cons(t_1',t_2')$

Then by definition of $\leq$, $t_1' \leq t_1$; $t_2' \leq t_2$.

Now $r(t') = cons(r(t_1'), r(t_2'))$

$r(t) = cons(r(t_1), r(t_2))$ so by induction

$r(t') \leq r(t)$ □

Lemma 9 Let Σ and ε be defined as above, and suppose that q is the congruence generated on CT_Σ by ε. Then for any $t \in CT_\Sigma$, there is a term $\bar{t} \in CT_\Sigma$ such that $\bar{t}$ is reduced and $(t,\bar{t}) \in q$.

Proof i) If t is finite, let $\bar{t} = r(t)$. By lemma 7, $\bar{t}$ is reduced and $(t,\bar{t}) \in q$.

ii) If t is infinite, then there is a directed set $<t_i>_{i\in I}$ such that $t = \sqcup_i t_i$ where each t_i is finite. Let $\bar{t} = \sqcup_i r(t_i)$, which exists since by lemma 8 r is monotonic and hence $<r(t_i)>_{i\in I}$ is directed. But for each $i \in I$, $(t_i, r(t_i)) \in q$, so by continuity of q $(t,\bar{t}) \in q$. Also by definition $\sqcup_i r(t_i)$ is reduced when each $r(t_i)$ is. Hence $\bar{t}$ is reduced. □

We can now show that L_Σ is indeed a normal term algebra for lists.

Theorem 10 L_Σ is a normal term algebra for lists.

Proof By construction the carrier of L_Σ is a subset of the carrier of CT_Σ. Furthermore it can be checked that L_Σ satisfies ε and is Δ-continuous. We must show that L_Σ is isomorphic to CT_Σ/q for q as defined in the previous lemma.

Let $g: L_\Sigma \to CT_\Sigma/q$

be defined as the restriction of the canonical mapping $\Theta : CT_\Sigma \to CT_\Sigma/q$ to L_Σ . Since L_Σ is a subalgebra of CT_Σ, g is a homomorphism.

By lemma 8, given any $t \in [t']$ for $[t'] \in CT_\Sigma/q$ we can reduce t to $\bar{t}$ so that $\bar{t} \in [t']$ and $\bar{t} \in L_\Sigma$. Thus $g(\bar{t}) = [t']$ and g is a surjection.

Assume $g(t_1) = g(t_2)$. Then $[t_1] = [t_2]$ and since L_Σ satisfies ε, we must have that $t_1 = t_2$ (by lemma 4). Thus g is an injection. g is thus the required isomorphism. □

We can now prove that CT_Σ/q is indeed initial in the class of Δ-continuous Σ-algebras satisfying ε.

<u>Theorem 11</u> CT_Σ/q is initial in the class of Δ-continuous Σ-algebras satisfying ε.

<u>Proof</u> Follows directly from theorem 10 and theorem 7. □

4. <u>Shared Structures</u>

A data structure can be viewed as a set of recursive equations in the program variables and some auxiliary variables. Sharing within the structure is reflected by the variables.

<u>Definition 8</u> A Σ-data structure is a function

$$e : X \to CT_\Sigma(X)$$

where X is a countable set whose elements are called variables. □

It was shown in chapter 2 that every such set of equations has a solution in CT_Σ, denoted $|e|$, which is a tuple

$$<|e|_{x_1}, |e|_{x_2}, \ldots >$$

where $|e|_{x_i}$ denotes the component of the solution corresponding to x_i. We then have that

$$\forall x_i \in X \quad |e|_{x_i} = e(x_i) \circ |e|$$

where $t \circ <t_1,t_2,\ldots >$ denotes the expression in CT_Σ obtained by simultaneously substituting each t_i for the corresponding x_i in t.

Definition 9 The list axioms of definition 4 are called the universal list axioms and are denoted E. □

The set of equations e defining a structure can be viewed as axioms on some nullaries of the list structures. Adding these equations to the universal axioms yields an algebra which may be viewed itself as the data structure. It includes as before, all equivalences induced by the universal axioms, but also includes those equivalences induced by the particular structure.

Our development depends heavily on the fact that $CT_\Sigma(X)$ is isomorphic to $CT_{\Sigma(X)}$ where, in the latter case, x in X is considered as a nullary or constant function of the algebra. Thus the equations in e can be said to characterise the "operations" x in X in exactly the same way that the equations in E characterise the elements of Σ . Thus, given $CT_\Sigma(X)/q_E$ the equations e create equivalences between some distinct congruence classes in $CT_\Sigma(X)/q_E$.

Intuitively, a data type is characterized by the algebra whose elements are congruence classes of terms which are equivalent because of the type axioms. Hence, for example, in type List,

$$hd(cons(t\ell\ x,b)) \equiv t\ell\ x,$$

regardless of the value of x . In addition, however, a particular structure will impose additional equivalences on terms if the structure involves sharing. Thus for example, if

$$x = cons(a,cons(y,b))$$

then

$$hd(hd(cons(t\ell\ x,b)) \equiv y,$$

a conclusion which can only be drawn with the knowledge of some particular structure. Note that these equivalences will change as the underlying structure is updated. The universal type axioms themselves must be true in any structure.

Given a congruence q, [t] is the set of all terms congruent to t in q. We will write $[t]_E$ to denote classes in $CT_{\Sigma(X)}/q_E$ where q_E is the congruence on $CT_{\Sigma(X)}$ generated by E, and $[t]_e$ to denote classes in $CT_{\Sigma(X)}/q_e$. Here q_e is the congruence on $CT_{\Sigma(X)}/q_E$ generated by e, or equivalently, q_e is the congruence on $CT_{\Sigma(X)}$ generated by e and E together.

The next theorem characterizes the quotient $CT_{\Sigma(X)}/q_e$. If two terms t_1, t_2 are equivalent in q_e then the terms $t_1 \circ |e|$ and $t_2 \circ |e|$ are equivalent in $CT_{\Sigma(X)}/q_E$. For example, let e be given by

$$x = cons(a,x).$$

Then we can show that

$$[x]_e = [t\ell^i x]_e$$

for any $i \in \omega$, since

$$[cons(a,cons(a,\dots))]_e = [t\ell^i(cons(a,cons(a,\dots))]_E$$

for any $i \in \omega$.

<u>**Theorem 12**</u> Let e be a list structure and let q_e be the congruence on $CT_{\Sigma(X)}$ generated by e together with E, and q_E be the congruence on $CT_{\Sigma(X)}$ generated by E.

Then

$$[t_1]_e = [t_2]_e \Rightarrow [t_1 \circ |e|]_E = [t_2 \circ |e|]_E.$$

<u>Proof</u> Let $s_e : CT_{\Sigma(X)}/q_E \to CT_{\Sigma(X)}/q_E$ be defined as

$$s_e([t]_E) = [t \circ |e|]_E.$$

Clearly s_e is a function, where

$$\ker(s_e) = \{<[t_1]_E, [t_2]_E> \mid s_e([t_1]_E) = s_e([t_2]_E)\}$$

Furthermore, $\ker(s_e)$ is an equivalence relation. We show that s_e is also a continuous congruence.

i) Let $<[t_i]_E, [t_i']_E> \in \mathrm{Ker}(s_e)$ $1 \le i \le n$ ________(*)

We must show that

$<\sigma([t_1]_E, [t_2]_E, \dots, [t_n]_E), \sigma([t_1']_E, \dots, [t_n']_E)> \in \ker(s_e)$

for each $\sigma \in \Sigma$.

Now

$s_e(\sigma([t_1]_E, \dots, [t_n]_E)) = s_e([\sigma t_1, \dots, t_n]_E)$ since q_E is a congruence.

$= [(\sigma t_1, \dots, t_n) \circ |e|]_E$ by definition of s_e.

$= [\sigma(t_1 \circ |e|, \dots, t_n \circ |e|)]_E$ by a property of $\circ$ if σ is not a nullary.

$= \sigma([t_1 \circ |e|]_E, \dots, [t_n \circ |e|]_E)$ by definition of σ in $CT_{\Sigma(X)}/q_E$.

$= \sigma(s_e([t_1]_E), \dots, s_e([t_n]_E))$ by definition of s_e.

$= \sigma(s_e[t_1']_E), \ldots, s_e([t_n']_E)$ by (*) .

$= [\sigma(t_1' \circ |e|, \ldots, t_n' \circ |e|)]$ by definition of σ and s_e .

$= [\sigma(t_1', \ldots, t_n') \circ |e|]_E$ by a property of $\circ$.

$= s_e([\sigma t_1', \ldots, t_n']_E)$ as required.

If σ is a nullary, the result is immediate.

ii) Suppose that $<t_i>_{i\in I}$ and $<\bar{t}_i>_{i\in I}$ are directed sets, and

$$<[t_i]_E, [\bar{t}_i]_E> \in \ker(s_e) \quad \text{for each } i \in I.$$

We must show that

$$<\sqcup_i[t_i]_E, \sqcup_i[\bar{t}_i]_E> \in \ker(s_e) .$$

Now

$s_e(\sqcup_i[t_i]_E) = s_e([\sqcup_i t_i]_E)$ by lemma 3.

$= [\sqcup_i(t_i \circ |e|)]_E$ by definition of s_e and continuity of $\circ$

$= \sqcup_i[t_i \circ |e|]_E$ by lemma 3.

$= \sqcup_i(s_e([t_i]_E))$ by definition of s_e.

$= \sqcup_i([\bar{t}_i \circ |e|]_E)$ by the hypothesis and by definition of s_e.

$= [(\sqcup_i\bar{t}_i) \circ |e|]_E$ by lemma 3 and continuity of $\circ$.

$= s_e([\sqcup_i\bar{t}_i]_E)$

$= s_e(\sqcup_i[\bar{t}_i]_E)$ as required.

iii) $\ker(s_e)$ contains e since

$s_e([x]_E) = [x \circ |e|]_E = [|e|_x]_E$ since $|e|$ is the solution to e,

and $s_e([e(x)]_E) = [e(x) \circ |e|]_E = [|e|_x]_E$ since $|e|$ is the solution to

e, as required.

Now by definition, q_e is the least continuous congruence on $CT_{\Sigma(X)}/q_E$ containing e, and so

$$q_e \subseteq \ker(s_e).$$

Thus if

$[t_1]_e = [t_2]_e$ then $<[t_1]_E,[t_2]_E> \in q_e$

so that $<[t_1]_E,[t_2]_E> \in \ker(s_e)$

and hence $[t_1 \circ |e|]_E = [t_2 \circ |e|]_E$ by definition of s_e. □

<u>Lemma 13</u> Let e, E, q_e and q_E be as in Theorem 12. Then if $t_1, t_2 \in CT_\Sigma$ (i.e. t_1 and t_2 do not contain variables) then

$$[t_1]_e = [t_2]_e \Rightarrow [t_1]_E = [t_2]_E .$$

<u>Proof</u> By Theorem 12

$$[t_1]_e = [t_2]_e \Rightarrow [t_1 \circ |e|]_E = [t_2 \circ |e|]_E.$$

But if $t \in CT_\Sigma$, then $t \circ |e| = t$, and hence

$[t_1]_E = [t_2]_E$ as required. □

The main result of this section can now be presented: each data structure e generates an algebra that is initial in the class of all continuous algebras satisfying both E and e.

<u>Theorem 14</u> Let e be a list structure and q_e be the congruence on $CT_{\Sigma(X)}$ generated by e and E. Then $CT_{\Sigma(X)}/q_e$, denoted by L_e, is initial in the class of all continuous $\Sigma(X)$-algebras satisfying E and e, together with continuous $\Sigma(X)$-homomorphisms between them. We denote this class by $\underline{\underline{\Delta Alg}}_{\Sigma,E,e}$.

<u>Proof</u> We must show that

$$nf_e : CT_{\Sigma(X)} \to CT_{\Sigma(X)}$$

exists and is a normalizer. Let

$$nf : CT_{\Sigma(X)} \to CT_{\Sigma(X)}$$

be the normalizer for $CT_{\Sigma(X)}/q_E$. Now define

$$nf_e(t) = nf(t \circ |e|), \quad t \in CT_{\Sigma(X)} \ .$$

i) Assume $[t_1]_e = [t_2]_e$

so $[t_1 \circ |e|]_e = [t_2 \circ |e|]_e$ by a simple property of congruences.

Then, by lemma 13

$$[t_1 \circ |e|]_E = [t_2 \circ |e|]_E$$

and so

$nf(t_1 \circ |e|) = nf(t_2 \circ |e|)$ since nf is a normalizer for q_E.

Hence by definition of nf_e,

$nf_e(t_1) = nf_e(t_2)$ as required.

ii) $[nf_e(t)]_e = [nf(t \circ |e|)]_e$ by definition of nf_e.

But

$[nf(t \circ |e|)]_E = [t \circ |e|]_E$ since nf is a normalizer for q_E.

Furthermore $q_E \subseteq q_e$ by minimality of q_E.

Thus $[nf(t \circ |e|)]_e = [t \circ |e|]_e$

$= [t]_e$ by a simple property of congruences.

Therefore $[nf_e(t)]_e = [t]_e$ as required.

iii) nf_e is clearly continuous since nf and $\circ$ are. □

If L_Σ is the canonical term Σ-algebra for E, we can view L_e as being the algebra obtained from L_Σ by "evaluating" the terms $t \in L_\Sigma$ by solving for e and then taking $t \circ |e|$. However, e itself must be retained when defining updates to a data structure, for the reasons discussed previously.

5. Updating List Structures

We must now indicate how updating list structures can be formalized in an algebraic setting. Clearly, if we modify some list structure e, we will also modify the algebra L_e by transforming it to some new algebra $L_{e'}$. We do this via a transformation to be defined below.

For a set of equations to reflect properly the sharing of some structure, we require that:

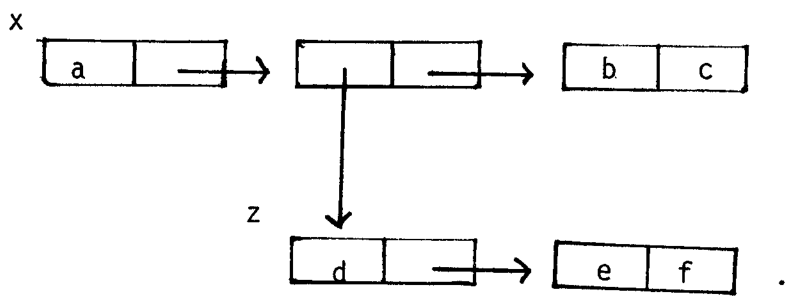
(1) Any shared sublist be identified by a variable;

(2) All references to a shared sublist must be through a variable naming that sublist.

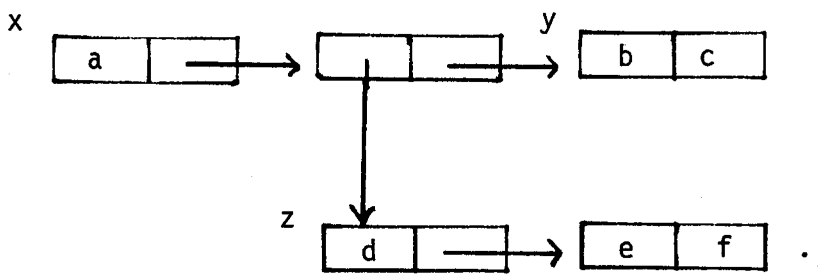
Consider the following list structure:

$$\begin{aligned} & x = \mathrm{cons}(a,\mathrm{cons}(z,\mathrm{cons}(b,c))) \\ (e) \qquad & z = \mathrm{cons}(d,\mathrm{cons}(e,f)) \\ & y = \perp \quad (\text{i.e. } y \text{ is undefined}) \end{aligned}$$

which may be represented diagramatically by:



Suppose that the updated structure is:

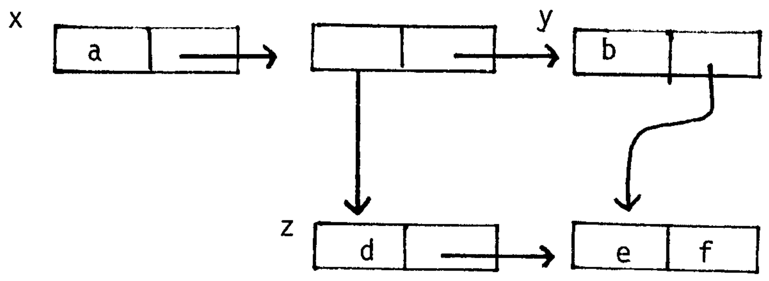


If we call the transformation rule R and use the convention R(u,v,e) to indicate the replacement of u by v in e, then we may express this transformation by R(y,*tℓtℓ* x,e). That is, we change the definition of y from whatever it was (in the above case it was undefined) to now be whatever expression is referenced by *tℓtℓ* x (which is cons(b,c)), and change the definition of x so that *tℓtℓ* x = y. Thus the new list structure is

```
            x = cons(a,cons(z,y))
(e')        z = cons(d,cons(e,f))
            y = cons(b,c)
```

Thus it is not enough simply to reset the value of y (by changing the third equation). The fact that *tℓtℓ* x (in the original list structure e) is now shared by x and y must be indicated in the equations.

Suppose that now we update e' to obtain:



This can be specified as $R(t\ell\, y, t\ell\, z, e')$ (or in fact by $R(t\ell t\ell t\ell\, x, t\ell\, z, e')$, etc.), which may be expressed as: Replace the structure pointed to by $t\ell\, y$ in e' (i.e. c) by the structure pointed to by $t\ell\, z$ (i.e. the structure cons(e,f)). In order to satisfy the two criteria above (1 and 2) we must introduce a new variable z' to indicate the fact that cons(e,f) is now shared by y and z. Thus we get

$$x = \text{cons}(a,\text{cons}(z,y))$$
$$y = \text{cons}(b,z')$$
$$z = \text{cons}(d,z')$$
$$z' = \text{cons}(e,f) \quad .$$

Note that a special role is played in the above examples by expressions of the form px for $p \in \{hd, t\ell\}^*$ and x a variable. Such expressions are called paths because they can be evaluated in a given list structure e (and reference some substructure). Thus they can be used to indicate the substructure to be replaced as in the above two examples.

The data type LIST is the many-sorted algebra with carrier

$<Path, L_\Sigma , L>$ sorted by P,L,A where:

i) $Path = \{hd, t\ell\}^* \cdot X$;

ii) $L_\Sigma \cong CT_\Sigma(X)/q_E$

iii) $L = \{ e | e$ is a list structure$\}$.

The set of operators in LIST is denoted by Ω and contains two operators:

i) Φ is of type $<\lambda,A>$ and $\Phi_{LIST} = e_\perp$ where $e_\perp = \{x = \perp | x \in X\}$. That is, Φ_{LIST} is the uninitialised list structure;

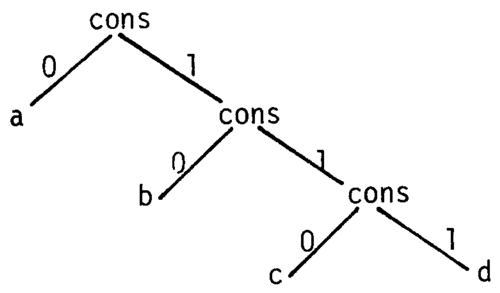
ii) R is of type $<PLA,A>$ and so

$$R_{LIST}: Path \times L_\Sigma \times L \to L.$$

R is defined by describing its effects on a set of equations e. Recall that elements $t \in CT_\Sigma(X)$ are partial functions

$$t: \omega^* \multimap \Sigma(X) .$$

Elements of ω^* for which t is defined are called <u>paths</u>. Thus, for example, the tree



could be represented by the partial function

$$t(\lambda) = t(1) = t(11) \mapsto \text{cons}$$

$$t(0) \mapsto a;\ T(10) \mapsto b;\ t(110) \mapsto c;\ t(111) \mapsto d.$$

We could, instead of labeling the edges of the tree with 0's and 1's, label them with hd and $t\ell$. Then we could define trees in terms of partial functions

$$t:\{hd,t\ell\}^* \multimap \Sigma(X).$$

This we do by associating hd with 0 and $t\ell$ with 1. To be consistent with list notation, we will <u>reverse</u> the string used to denote a path in the tree. Thus in the above example, we will have

$$t(\lambda) = t(t\ell) = t(t\ell t\ell) \mapsto \text{cons}$$

$$t(hd) \mapsto a;\ t(hdt\ell) \mapsto b;\ t(hdt\ell t\ell) \mapsto c;$$

$$t(t\ell t\ell t\ell) \mapsto d.$$

Formally then

$$t:\{hd,t\ell\}^* \multimap\!\!\rightarrow \Sigma(X)$$

is defined in terms of

$$t:\{0,1\}^* \multimap\!\!\rightarrow \Sigma(X)$$

as follows: If $p \in \{hd,t\ell\}^*$, $w \in \{0,1\}^*$, then

$$t(p) = \sigma \iff t(w) = \sigma$$

where, if $p = \delta_1\delta_2\ldots\delta_n$, $w = i_1i_2\ldots i_n$ and

$$i_j = \begin{cases} 0 & \text{if } \delta_j = hd \\ 1 & \text{if } \delta_j = t\ell \end{cases} .$$

<u>Definition 10</u> Let e be a function

$$e:X \rightarrow CT_\Sigma(X) .$$

If $p \in \{hd,t\ell\}^*$, $x \in X$ and $e(x)$ is denoted t_x, then

$$ref(px) = \begin{cases} \langle p,x\rangle & \text{if } t_x(p) \text{ is defined} \\ ref(\nu y) & \text{if } \exists u,\nu \in \{hd,t\ell\}^*,\ \nu \neq \lambda , \\ & \quad \nu u = p \text{ and } t_x(u) = y \\ \perp & \text{otherwise .} \end{cases}$$

Intuitively, px represents a way of "traversing" the structure e. If $\mathrm{ref}(px) = \langle q,y\rangle$, then px references a subexpression of $e(y)$ specified by q. For example, if

$$x = \mathrm{cons}(a,\mathrm{cons}(b,y))$$
$$y = \mathrm{cons}(a,\mathrm{cons}(c,d))$$

then $\mathrm{ref}(t\ell t\ell t\ell\ x) = \langle t\ell,y\rangle$. The path $t\ell t\ell t\ell\ x$ references the subtree at the $t\ell$ of the value of y.

We now define the following substitution function: Let $x,y,m \in V$,

$$p,q,u,w \in \{hd,t\ell\}^*, \quad t \in L_\Sigma .$$

1) $t_x[y/m] = t'$ where

$$t'_x(u) = \begin{cases} m & \text{if } t_x(u) = y \\ t_x(u) & \text{otherwise.} \end{cases}$$

Each occurrence of y is replaced by m.

For example,

$$\mathrm{cons}(a,\mathrm{cons}(x,\mathrm{cons}\ (x,y)))[x/z] = \mathrm{cons}(a,\mathrm{cons}(z,\mathrm{cons}(z,y))).$$

2) $t_x[p/m] = t'$ where

$$t'(u) = \begin{cases} t_x(u) & \text{if } p \text{ is not a suffix of } u \\ m & \text{if } u = p \\ \perp & \text{otherwise.} \end{cases}$$

The subtree referenced by p is replaced by the variable m.

For example

$$\text{cons}(a,\text{cons}(x,\text{cons}(x,y)))[t\ell/z] = \text{cons}(a,z).$$

3) $t_x[p/t] = t'$ where

$$t'(u) = \begin{cases} t_x(u) & \text{if } p \text{ is not a suffix of } u. \\ t(w) & \text{if } u = wp \\ \perp & \text{if } u = wp,\ t = \perp . \end{cases}$$

The subtree referenced by p is replaced with subtree t.

For example

$$\text{cons}(a,\text{cons}(x,\text{cons}(x,y)))[t\ell/\text{cons}(a,b)] = \text{cons}(a,\text{cons}(a,b)).$$

4) Let $\text{ref}(py) = \langle q,z\rangle$, $p \in \{hd,t\ell\}^+$.

then
$$t_x[py/t] = \begin{cases} t_x & \text{if } x \neq z \\ t_x[q/t] & \text{if } x = z \end{cases}$$

If $\text{ref}(py) = \perp$, then $t_x[py/t] = \perp$.

If py references a subtree of t, replace this subtree with t.

Note that this substitution depends on e.

For example, if

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

$$y = \text{cons}(c,x)$$

then

$$\text{cons}(a,\text{cons}(b,c))[t\ell t\ell\ y/b] = \text{cons}(a,b).$$

A reference may also be evaluated (dereferenced) to yield a subtree.

$e(px) = t'$ where $ref(px) = <q,y>$ and

$t'(u) = t_x(qu)$. □

The substitution rule R is now defined in terms of these substitutions. $R(px,t,e)$ can be thought of as corresponding to the Algol-like statement

$px \leftarrow t$.

<u>Definition 11</u>

$R:\text{Path} \times L_\Sigma \times L \to L$

is defined by specifying

$R(p,t,e) = e'$

in terms of e'.

Then $R(x,y,e) = e'$ where e' is defined as follows:

1)
$$e'(z) = \begin{cases} y & \text{if } z = x \\ t_z[x/y] & \text{if } z \neq x \text{ .} \end{cases}$$

2) $R(x,\text{cons}(t_1,t_2),e) = e'$ where

$$e'(y) = \begin{cases} \text{cons}(t_1,t_2) & \text{if } y = x. \\ t_y[x/m] & \text{if } y \neq x,\ y \neq m \text{ and } m \text{ is new.} \\ t_x[x/m] & \text{if } y = m. \end{cases}$$

The chosen semantic effect of this assignemnt is to "move" the variable x to refer to a new list. Any old reference to x must still reference the old list, and this is why the variable m is introduced.

3) $R(x,py,e) = e'$ where

$$e'(z) = \begin{cases} e(py)[x/m] & z = x, \quad m \text{ is new.} \\ t_z[x/m][py/x] & z \neq x, \quad z \neq m. \\ t_x[x/m][py/x] & z = m. \end{cases}$$

The variable x is moved to the tree referenced by py. All occurences of x in e must be changed to m in e', since the list referred to by x or through x in e is not changed. Secondly, the subtree referenced by py must be replaced by x in accordance with the requirement that sublists be named. Note that this reference may appear in t_x or any other right hand side in e.

4) $R(px,\mathrm{cons}(t_1,t_2),e) = e'$ where

$$e'(y) = t_y[px/\mathrm{cons}(t_1,t_2)]$$

5) $R(px,qy,e) = e'$ where

$$e'(z) = \begin{cases} t_z[qy/m][px/m] & z \neq m, \quad \text{new } m \\ e(qy)[px/m] & z = m. \end{cases}$$

A sublist $e(qy)$ is now referenced from px, so it must be named by m.

6) $R(px,y,e) = e'$ where
$e'(z) = t_z[px/y]$.

7) $R(\bot,t, e) = e_\bot$

8) $R(p,\bot, e) = e_\bot$ □

<u>Example</u> We illustrate each case of the above rule.

1) Let $e = \phi$. (that is, $e(x) = \bot$ for each $x \in X$).

i) $R(x,cons(a,cons(b,c)), e)$ yields e' where

$x = cons(a,cons(b,c))$

by rule 2.

ii) $R(y,cons(a,cons(b,c)), e_1)$ yields e_2 ; where

$x = cons(a,cons(b,c))$

$y = cons(a,cons(b,c))$

iii) $R(t\ell\ x,y, e)$ yields

$x = cons(a,y)$

$y = cons(a,cons(b,c))$

iv) $R(z,y, e_2) = e_3$ where

$x = cons(a,y)$

$y = cons(a,cons(b,c))$

v) $R(y, cons(a,b), e_3) = e_4$ where

$y = cons(a,b)$

$x = cons(a,m)$

$z = m$

$m = cons(a, cons(b,c))$

vi) $R(t\ell\ z, z, e_5) = e_6$ where

$y = cons(a,b)$

$x = cons(a,m)$

$z = m$

$m = cons(a,z)$

vii) $R(t\ell\ z, t\ell\ x, e_6) = e_7$

Note that $ref(t\ell\ z) = \langle t\ell, m\rangle$

$ref(t\ell\ x) = \langle t\ell, x\rangle$

So e_z is

$y = cons(a,b)$

$x = cons(a,m')$

$z = m$

$m = cons(a,m')$

$m' = m$

Whenever equations of the form

$$x = m \qquad (\text{or} \quad m = x)$$

appear, m new, then every occurence of m can be replaced by x, and the equation

$$m = t$$

can be replaced by

$$x = t \ .$$

(In the second case, the equation in m can be eliminated.)

Hence in the above example we would have

$$y = \text{cons}(a,b)$$
$$x = \text{cons}(a,z)$$
$$z = \text{cons}(a,z).$$

2) Let e be

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

Then $R(t\ell\ x, t\ell t\ell\ x, e) = e'$ where e' is

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

since $(\text{cons}(a,\text{cons}(b,\text{cons}(c,d)))[t\ell\ x/m])[t\ell t\ell\ x/m]$

$= (\text{cons}(a,m))[t\ell t\ell\ x/m] = \text{cons}(a,m)$

With the same e,

$$x = \mathrm{cons}(a,\mathrm{cons}(b,\mathrm{cons}(c,d)))$$

consider

$$R(t\ell t\ell\ x, t\ell\ x, e):$$

$$x = \mathrm{cons}(a,m)$$

$$m = \mathrm{cons}(b,m)$$

since $(\mathrm{cons}(a,\mathrm{cons}(b,\mathrm{cons}(c,d))))[t\ell t\ell\ x/m])[t\ell\ x/m]$

$= (\mathrm{cons}(a,\mathrm{cons}(b,m)))[t\ell\ x/m]$

$= \mathrm{cons}(a,m)$

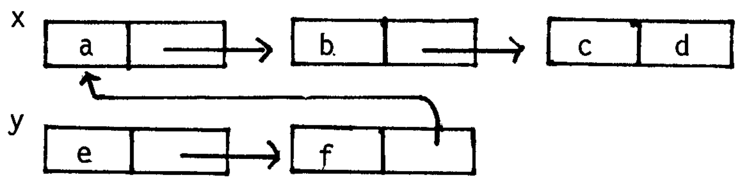
and $\mathrm{cons}(b,\mathrm{cons}(c,d))[t\ell t\ell\ x/m] = \mathrm{cons}(b,m)$. □

A set of equations e can be visualized as a "box and arrow" diagram where each box corresponds to the operator cons, and each arrow corresponds either to a nested expression or a variable.
For example, the set of equations

$$x = \mathrm{cons}(a,\mathrm{cons}(b,\mathrm{cons}(c,d)))$$

$$y = \mathrm{cons}(e,\mathrm{cons}(f,x))$$

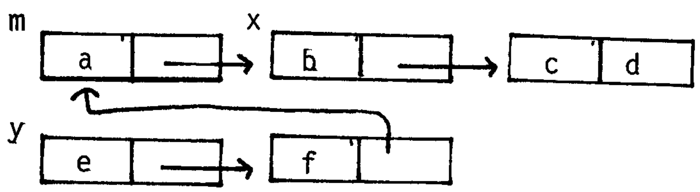
has corresponding diagram



The operator R may be defined in terms of these diagrams in an obvious way. For example, consider the expression

$$R(x, t\ell\ x, e)\ .$$

In terms of diagrams, $R(x,py,e)$ changes the diagram so that x becomes associated with the box py. In addition, we give the old box associated with x a new name, m say.



The corresponding effect on equations is given by the rule

$R(x,py, e) = e'$ where

$$e'(z) = \begin{cases} e(py)[x/m] & z = x \\ t_z[x/m][py/x] & z \neq x,\ \ z \neq m \\ t_x[x/m][py/x] & z = m \end{cases}$$

which can be explained in terms of diagrams as follows:

i) $e'(x) = e(py)[x/m]$.

x is "moved" to the box pointed to by py. In terms of equations, any occurence of x in the old equations must be changed, because arrows to the old box labelled by x still point to the same box, which is now given the new name m.

ii) $\quad e'(m) = t_x[x/m][py/x]$

m labels the box that was labelled by x. In terms of equations, if py is a sublist of t_x, then this sublist must be replaced by x, so that the property that sublists be named is preserved.

iii) $\quad e'(z) = t_z[x/m][py/x]$ $\quad$ for all other variables z.

The reasons for the two substitutions here are the same as those discussed above.

It is possible to check informally the definition of each case of R in a similar way to that given above. In addition it is fairly easy to see that R preserves the criteria 1) and 2) above that shared sublists be identified and that references to shared sublists are through a variable naming the sublist.

It can be shown that LIST is a continuous algebra. To do this we must show that the carriers of LIST are complete partial orders and that R is continuous with respect to this order.

Path can be made into a (flat) cpo by adding $\perp_p$ and defining $p \le_L p'$ if and only if $p = \perp_p$ or $p = p'$. As for L_Σ, it is already a cpo. We can order L as follows:

$e \le_L e'$, $\quad$ if and only if $e \le_{CT} e'$. That is, if for all $x \in X$, $e(x) \le_{CT} e'(x)$.

□

<u>Lemma 15</u> $\leq_L$ is a partial order on L and is complete.

<u>Proof</u> a) $\leq_L$ is a partial order

i) $\forall x \; e(x) \leq_{CT} e(x)$

$\therefore \; e \leq_L e$ by definition of $\leq_L$.

ii) Suppose that

$e_1 \leq_L e_2$ and $e_2 \leq_L e_3$

Then by definition of $\leq_L$

$\forall x. \; e_1(x) \leq_{CT} e_2(x)$ and $e_2(x) \leq_{CT} e_3(x)$

so $\forall x \; e_1(x) \leq_{CT} e_3(x)$ since $\leq_{CT}$ is a partial order

hence $e_1 \leq_L e_3$.

iii) Suppose $e_1 \leq_L e_2$ and $e_2 \leq_L e_1$

then

$\forall x \; e_1(x) \leq_{CT} e_2(x), \; e_2(x) \leq_{CT} e_1(x)$

so $\forall x \; e_1(x) = e_2(x)$ since $\leq_{CT}$ is a partial order

hence $e_1 = e_2$.

b) $\leq_L$ is complete.

Let $< e_i >_{i \in I}$ be a directed set in L.

Then by definition of $\leq_L$, $<e_i(x)>_{i \in I}$ is directed for every $x \in X$.

Let $<e>$ denote the tuple which contains the least upper bound in $CT_{\Sigma(X)}$ of each directed set $<e_i(x)>$. $<e>$ exists by completeness of $CT_{\Sigma(X)}$.

Now let

$$e:X \to CT_{\Sigma}(X)$$

be defined by

$$e(x) = <e>_x$$

Then for all e and for all x,

$$e_i(x) \leq_{CT} e(x) .$$

Thus e is an upper bound of $< e_i >$.

Suppose however that e' was another upper bound.

Then for all i and for all x,

$$e_i(x) \leq_{CT} e'(x)$$

hence for all x,

$e(x) \leq e'(x)$ since $<e>_x$ is the least upper bound of $<e_i(x)>$.

Thus e is the least upper bound of $< e_i >$. □

<u>Lemma 16</u> If $<e_i(z)>_{i \in I}$ are directed for each $z \in X$, and

$e(z) = \sqcup_i \{e_i(z)\}$, then

$<e_i(z)[x/y]>$ is directed for each $x,y,z \in X$, and

$e(z)[x/y] = \sqcup_i <e_i(z)[x/y]>$.

<u>Proof</u> Let $<e_i(z)>$ be a directed set with $z \in X$. Now let $u \in \omega^*$.

$$(e_i(z)[x/y])(u) = \begin{cases} y & \text{if} \quad e_i(z)(u) = x \\ e_i(z)(u) & \text{otherwise.} \end{cases}$$

But if $e_j(z)(u) = x$ for some j then by definition of directes sets $e_k(z)(u) = x$ for all $k \geq j$. Furthermore, since $e(z) = \sqcup_i \{e_i(z)\}$, $e(z)(u) = x$.

Hence $<e_i(z)(x/y)>$ is directed with $e(z)(x/y)$ as upper bound. □

<u>Lemma 17</u> If $<e_i(x)>$ is a directed set for each $x \in X$, where

$e(x) = \sqcup_i \{e_i(x)\}$

then

$<e_i(x)[\rho y/t]>$

is directed for each x, and

$$e(x)[py/t] = \sqcup_i\{e_i(x)[py/t]\}.$$

Proof Let $\langle e_i(x)\rangle$ be a directes set. Then consider ref(py) for each e_i, which we denote $\mathrm{ref}_i(py)$.
Suppose that for some j,

$$\mathrm{ref}_j(py) = \langle q,z\rangle$$

is defined. Then

$$e_j(z)(q) = \sigma$$

say must be defined from some σ. But then for any $k \geq j$

$$e_k(z)(q) = \sigma .$$

Also, there must be some sequence

$$\mathrm{ref}_j(v_0x_0) = \mathrm{ref}_j(v_1x_1) = \mathrm{ref}_j(v_2x_2) = \dots = \mathrm{ref}_j(v_nx_n)$$
$$= \langle q,z\rangle$$

by definition of z, where $v_0x_0 = py$, $v_nx_n = qz$ and

$$v_0 = v_1u_1,\ v_1 = v_2u_2,\dots,v_{n-1} = v_nu_n$$

and such that $e_j(x_i)(u_i) = \sigma_i$ for some $\sigma_i \in \Sigma$. If this were not the case, ref(py) would be undefined. Now since $\langle e_i(x)\rangle$ is directed, $e_j(x) = \sigma \Rightarrow e_k(x) = \sigma$ for all $k \geq i$.

Hence $\langle ref_i(py) \rangle$ is a chain of the form

$$\begin{tikzcd}
\langle q,z \rangle \arrow[d, dash] \\
\perp
\end{tikzcd}$$

and $ref(py) = \sqcup_i \{ref_i(py)\}$.

Now if $ref_j(py) = \perp$, then

$e_j(x)[py/t] = \perp$

If $\sqcup_i \{ref_i(py)\} = \perp$, then

$e(x)[py/t] = \perp$

and the result follows.

Otherwise, let $\langle q,z \rangle = ref(py) = \sqcup_i \{ref_j(py)\}$.

Then

$$e_i(x)[py/t] = \begin{cases} e_i(x) & \text{if } x \neq z \\ e_i(x)[q/t] & \text{if } x = z. \end{cases}$$

Then $\langle e_i(z)[q/t] \rangle$ is clearly directed since if $e_i(z)(q)$ is defined, then $e_k(z)(q) = e_i(z)(q)$ for all $k \geq i$. Also

$\langle e_i(x)[py/t]\rangle = \langle e_i(x)\rangle$ for all $x \in X$, $x \neq z$

is directed by supposition,

so $\langle e_i(y)[py/t]\rangle$ $x \in X$ is directed as required.

Also, for each $x \in X$, $x \neq z$

$\sqcup_i\{e_i(x)[py/t]\} = \sqcup_i\{e_i(x)\} = e(x)$ as required

and $\sqcup_i\{e_i(z)[py/t]\} = \sqcup_i\{e_i(z)[q/t]\}$

$= e(z)[q/t]$ since $\langle e_i(z)\rangle$ is directed and $\sqcup_i\{e_i(z)\} = e(z)$

$= e(z)[py/t]$ since $\mathrm{ref}(py) = \langle q,z\rangle$.

Hence for all $x \in X$

$e(x)[py/t] = \sqcup_i\{e_i(x)[py/t]\}$ as required. □

Lemma 18 If $\langle t_i\rangle$ is directed and $t = \sqcup_i t_i$, then

$\langle e(z)[px/t_i]\rangle$ is directed and

$\sqcup_i\{e(z)[px/t_i]\} = e(z)[px/t]$.

Proof Let $\mathrm{ref}(px) = \langle q,z\rangle$. (If $\mathrm{ref}(px) = \perp$ there is nothing to prove). Then

$$e(x)[px/t_i] = \begin{cases} e(x) & x \neq z \\ e(x)[q/t_i] & \text{if } x = z. \end{cases}$$

Hence to establish the result, we need only show that for any $\bar{t} \in L_e$,

$$\sqcup_i\{\bar{t}[q/t_i]\} = \bar{t}[q/t].$$

If $\bar{t}(q) = \bot$, then $\bar{t}[q/t] = \bot$ for any t, and the result follows. If $\bar{t}(q)$ is defined, then $\bar{t}[q/t_i]$ is the tree obtained from $\bar{t}$ by replacing the subtree at q by t_i, and so the result clearly follows.

□

Lemma 19 R_{LIST} is continuous

i) First argument:

$$R(\sqcup_i\{t_i\},t, e) = \sqcup_i R(t_i,t, e) .$$

Continuity follows from case 7 to definition 10.

ii) Second argument

(a) If the second argument is a path, then continuity follows from case 8 of definition 10.

(b) We must show that

$$R(t,\sqcup_i\{t_i\}, e) = \sqcup_i R(t,t_i, e)$$

where $\{t_i\}$ is directed.

Firstly, suppose that

$$\sqcup_i\{t_i\} = x .$$

Then for each i, $t_i = \bot$ or $t_i = x$, and if $t_j = x$ then $t_k = x$ for all $k \geq j$.

But by case 8 of definition 10

$$R(t,\bot,e) = e_\bot \qquad \text{for any } e.$$

But $e_\bot \leq e$ for any e, so

$$\{R(t,t_i,e)\}$$

forms a chain with upper bound $R(t,x,e)$ as required.

Secondly, suppose that

$$\sqcup_i\{t_i\} = px$$

for some $p \in \{hd,t\ell\}^*$. Then $\{t_i\}$ must be a chain of the form

$$\begin{tikzcd} px \arrow[d, dash] \\ \bot \end{tikzcd}$$

and by case 8 of definition 10,

$$R(t,\bot,e) = e_\bot$$

for any e, so

$$\{R(t,t_i,e)\}$$

forms a chain with upper bound $R(t,px,e)$ as required.

Finally, suppose that $\sqcup_i\{t_i\} = \text{cons}(t_1,t_2)$

Then if $t = x$,

$$e_i'(y) = e_i(y)[x/m] \quad \forall y \in X, \quad y \neq x, \quad y \neq m$$

and by lemma 16 $\quad \sqcup_i\{e_i'(y)\} = e(y)[x/m]$.

But e' (obtained from $R(t,\text{cons}(t_1,t_2),L_e)$) yields from case 4

$$e'(y) = e(y)[x/m]$$

as required. Also

$e'(m) = e(x)[x/m] = \sqcup_i\{e_i(x)[x/m]\}$ by lemma 16.

and $\quad e'(x) = \text{cons}(t_1,t_2) = \sqcup_i t_i$ by the supposition.

Hence $\quad R(x,\text{cons}(t_1,t_2),e) = \sqcup_i R(x,t_i,e)$.

If $t = px$, $p \neq \lambda$ and $\sqcup_i\{t_i\} = \text{cons}(t_1,t_2)$, then for each i

$$e'(y) = e(y)[px/t_i]$$

and by lemma 18 $\quad \sqcup\{e'(y)\} = e'(y) = e(y)[px/t]$ as required.

iii) Third argument: We must show that

$$R(p,t,\sqcup_i\{e_i\}) = \sqcup_i R(p,t,e_i)\ .$$

Consider each case of definition 10.

1) Note that if $<e_i>$ is directed, then $<e_i(z)>$ is directed for each $z \in X$.

Now

$$e'_i(z) = \begin{cases} y & \text{if } z = x \\ e_i(z)[x/y] & \text{if } z \neq x \end{cases}$$

But

$$e'(z) = \begin{cases} y & \text{if } z = x \\ e(z)[x/y] = \sqcup_i\{e_i(z)[x/y]\} & \text{by lemma 16} \end{cases}$$

and the result thus follows.

2) $R(x,\text{cons}(t_1,t_2),L_e)$

$$e'_i(z) = \begin{cases} \text{cons}(t_1,t_2) & \text{if } z = x \\ e_i(y)[x/m] & \text{if } z \neq x,\ z \neq m. \\ e_i(x)[x/m] & \text{if } z = m. \end{cases}$$

$$e'(z) = \begin{cases} \text{cons}(t_1,t_2) & \text{if } z = x \\ e(z)[x/m] = \sqcup_i\{e_i(z)[x/m]\} & \text{by lemma 14} \\ e(x)[x/m] = \sqcup_i\{e_i(x)[x/m]\} & \text{by lemma 14} \end{cases}$$

and hence the result follows.

3) - 6) All follow similarly by application of lemmas 16 or 17.

7) $R(\perp,t,e) = e_\perp$ for any e.

Hence contunity follows trivially

8) Follows trivially as well. □

We thus have establish that R is continuous. Since Φ is a nullary it is also continuous, and hence

Theorem 20 LIST is a Δ-continuous Ω-algebra. □

6. Conclusions

The idea of regarding data types as many-sorted algebras characterized by an initial quotient algebra has been extended to include types with possibly infinite values. It has been shown that the concepts of data sharing and circularity can be treated in this framework, using the general type LIST which allows arbitrary assignments rather than just the restricted assignment of say Pure LISP [32]. The characterization of types as initial quotient algebras has been to form congruence classes containing all expressions which, in accordance with some type axioms, evaluate equally. We have shown in this chapter that a data structure with possible sharing can be characterized as the initial quotient where terms may be congruent because of the type axioms, but they may also be equivalent because of the additional structure induced by the data structure. The type itself is then viewed as the collection of all such structures.

In chapter 5 it is shown how the formulation of lists with sharing in this chapter can be used in a simple program proof.

Chapter 5

Some Applications and Implications

1 Introduction

Two approaches to formalizing data types with sharing have been defined in chapters 3 and 4. We call the model in chapter 3 the reference model and the one in chapter 4 the continuous model.

The reference model is concerned primarily with describing the semantics of references, rather than general properties of sharing. The continuous model for lists, for example, could be implemented in the reference model. The continuous model, although illustrated mainly for lists, can be used to describe many types with sharing, and is easily used to prove properties of recursive programs using shared types. The reference model describes particular "lower level" concepts (such as in place) but it can be used for the verification of while programs using the type list. The programs most amenable to verification with the reference model are programs concerned more with references than sharing, such as "in place" reversal or marking programs.

The continuous model provides a setting for treating structures with sharing as a continuous algebra and thus the usual induction rules for continuous functions can be used to prove correctness of programs using the type. This is illustrated in section 3 of this chapter. The reference model was motivated in part by a desire to verify programs using the invariant assertion method (Floyd [15], Hoare [19]), and we illustrate how the proof rule presented in chapter 3 can be used together with the type axioms for verification. Both models do provide a language for expressing properties of data structures; certain properties, namely those particular to concepts of referencing (such as in place, equality of references) are naturally more easily expressed in a language based on the reference model. For both models it would be interesting to study formal systems to be used in proving theorems about programs using the types.

2 Reasoning with the Reference Model

Hoare's[19] original proof rule for "simple" assignment is

$$Q^x_e \ \{x \leftarrow e\}\ Q$$

where Q is an assertion and Q^x_e is Q with each free occurence of x replaced by e.

As discussed in chapter 3, generalized assignment in the reference model is viewed as assignment of an entirely new structure. That is, an assignment

$$p \leftarrow c$$

is viewed as

$$\sigma' \leftarrow U(\sigma,p,c)$$

where U is a function that returns a new structure, the structure which is identical to σ except that p is changed to c. In fact

$$U(\sigma,p,c) \text{ is just the structure } \leftarrow(p,c,\sigma).$$

Thus the proof rule is

$$Q^{\sigma}_{\leftarrow(p,c,\sigma)} \ \{p \leftarrow c\}\ Q$$

We will write σ' to denote $\leftarrow(p,c,\sigma)$ provided that p and c can be determined from the context of the assertion.

The type defined in chapter 3 is more general than the type "usually" called linked list. In the "usual" semantics of Algol-like languages (Scott-Strachey[38], Reynolds [35]), statements are viewed as mappings from one state to another

$$s: \; S \to S$$

Statement composition is then viewed as function composition:

$$S[\![s_1 ; s_2]\!](\rho) \overset{\Delta}{=} S[\![s_2]\!](S[\![s_1]\!](\rho))$$

We view an element $\sigma \in$ Struct as being part of a state, say S = Struct + S' (+ is disjoint union). Then the semantics of an assignment statement of the form

$$p \leftarrow \ell$$

can be defined as

$$S[\![p \leftarrow \ell]\!](\sigma + \rho) = \leftarrow(p,\ell,\sigma) + \rho$$

Clearly then

$$\begin{aligned} S[\![p_1 \leftarrow \ell_1 ;\; p_2 \leftarrow \ell_2]\!](\sigma + p) \\ &= S[\![p_2 \leftarrow \ell_2]\!](S[\![p_1 \leftarrow \ell_1]\!](\sigma + \rho)) \\ &= S[\![p_2 \leftarrow \ell_2]\!](\leftarrow(p_1,\ell_1,\sigma) + \rho) \\ &= \leftarrow(p_2,\ell_2, \leftarrow(p_1,\ell_1,\sigma)) + \rho \end{aligned}$$

Thus a program segment of the form

$$p_1 \leftarrow \ell_1;$$
$$p_2 \leftarrow \ell_2;$$
$$\vdots$$
$$p_n \leftarrow \ell_n;$$

can be viewed as the expression

$$\leftarrow(p_n, \ell_n, \leftarrow(p_{n-1}, \ell_{n-1}, \leftarrow(\ldots \leftarrow(p_1, \ell_1, \sigma))\ldots)$$

where σ is the "initial structure" of the segment.
To state assertions about list structures we use a language which describes terms in T_Σ, and which has the usual logical symbols, as well as equality (=). The interpretation of a term $t \in T_\Sigma$ is the class $[t] \in T_\Sigma/q$ where q is the last congruence defined by the type axioms given in Chapter 3. We will typically quantify over terms $t \in T_\Sigma$. This is illustrated below with some examples of the use of the proof rule.

<u>Example 1</u> Consider the statement

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

and suppose the postcondition is

$$Q:\ \ t\ell t\ell\ y(\sigma) = x(\sigma)\ \&\ z(\sigma) \neq y(\sigma)$$

Then

$$Q^{\sigma}_{\sigma'}:\ \ t\ell t\ell\ y(\sigma') = x(\sigma')\ \&\ z(\sigma') \neq y(\sigma')$$

Now, applying axiom 9 we have

$t\ell t\ell\ y(\sigma') = t\ell(t\ell(y(\sigma'))(\sigma'))(\sigma')$ (Notation)

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

Now $z(\sigma') = z(\sigma)$ and $x(\sigma') = x(\sigma)$ by axiom 5 so

$$t\ell(y(\sigma))(\sigma') = z(\sigma) \qquad \text{by axiom 9}$$

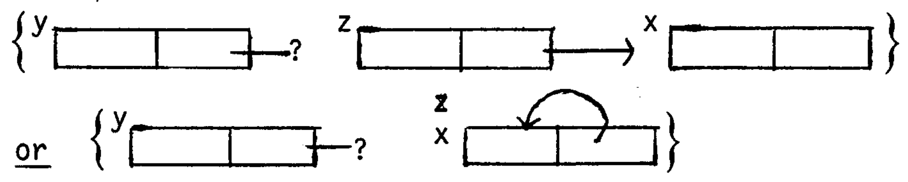
where $p_1(\sigma)$ is $y(\sigma)$ and p_2 is y and hence $p_1(\sigma) = p_2(\sigma)$. Hence

$$\begin{aligned} t\ell(t\ell(y(\sigma'))(\sigma'))(\sigma') &= t\ell(z(\sigma))(\sigma') \\ &= t\ell\ z(\sigma) \quad \text{since } z(\sigma) \neq y(\sigma) \\ &\quad \text{and by notation.} \end{aligned}$$

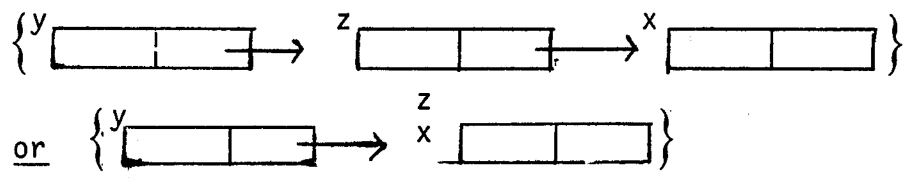
Hence $Q^{\sigma}_{\sigma'}$ reduces to

$$t\ell\ z(\sigma) = x(\sigma) \ \ \& \ \ z(\sigma) \neq y(\sigma)$$

Pictorially, we have



$t\ell\ y \leftarrow z$



If the post-condition includes

$$z(\sigma) \neq x(\sigma)$$

(the top diagram), then the precondition would include the same clause (by axiom 5), again implying the top diagram of the precondition.

In general an assertion about a structure can be satisfied by a number of different structures; in a proof the assertion must restrict the number of possible structures sufficiently to deduce the required assertions. For example, if the clause

$$z(\sigma) \neq y(\sigma)$$

was omitted from the post-condition, then we would have as pre-condition

$$t\ell\ z(\sigma) = x(\sigma) \quad \text{or} \quad z(\sigma) = x(\sigma).$$

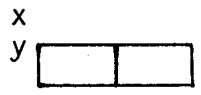
2. Consider the statement

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

with post-condition

$$Q:\ \ y(\sigma) = x(\sigma).$$

Pictorially the post-condition is



Then $\eta^{\sigma}_{\sigma'}$: $y(\sigma') = x(\sigma')$

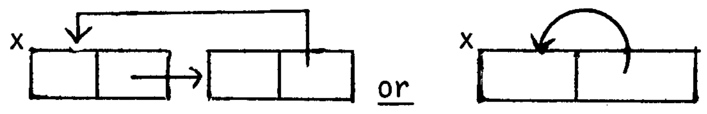
but $x(\sigma') = x(\sigma)$ axiom 6

$y(\sigma') = \mathit{tltl}\ x(\sigma)$.

Thus the precondition is

$\mathit{tltl}\ x(\sigma) = x(\sigma)$

which is pictorially



Intuitively, if the assignment statement causes y and x to refer to the same cell, after it is executed, then $\mathit{tltl}\ x$ must have refered to the same cell as x before the assignment.

3. Consider the statement

$y \leftarrow \mathrm{cons}(a,\mathrm{cons}(b,x))$

with post-condition

$\mathit{tltl}\ y(\sigma) = z(\sigma)$.

Then the precondition is

$Q^{\sigma}_{\sigma'}$: $\mathit{tltl}\ y(\sigma') = z(\sigma')$

where

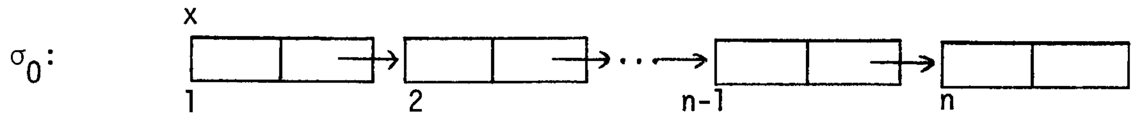
$\mathit{tltl}\ y(\sigma') = x(\sigma)$	Axiom 6
$z(\sigma') = z(\sigma)$	Axiom 6

so the precondition reduces to

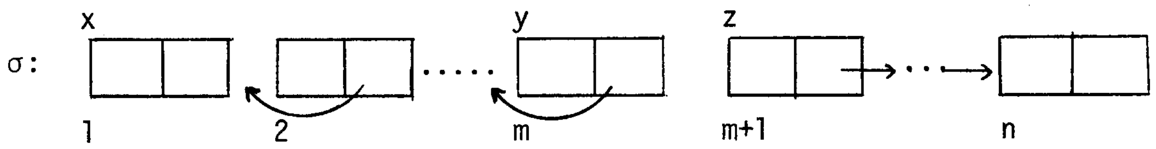
$$x(\sigma) = z(\sigma). \qquad \square$$

The generality of the type definition allows for the expression of more powerful conditions than were illustrated in the above examples. It is possible to relate values in a structure to the values held in the structure after the structure has been updated.

For example, suppose σ_0 is a structure represented diagramatically as follows:



Suppose that at some stage in the execution of a program, σ_0 has been updated to σ, which is represented as:



We have associated an index with each cell because the diagram is insufficient to capture certain aspects of the structure. For example, in σ we wish to imply that z and y are in some sense adjacent. That is, the cell referred to by z in σ was referred by the $t\ell$ in σ_0 of the cell y refers to in σ. That is

$$t\ell(y(\sigma))(\sigma_0) = z(\sigma).$$

With a less abstract underlying semantics, it would be necessary to identify the cells (as we have in the diagram) and argue that if $id(y) = x_1$, $id(z) = x_2$ in σ, then in σ_0

$$t\ell(x_1) = x_2$$

The necessity for this type of identification, we believe, leads to a more notationally complex statement of assertions as well as more complex proof rules.

The generality of the type linked-list defined in chapter 3, together with the treatment of references, allows for a fairly direct way to state assertions of programs manipulating the type. This is illustrated with a simple program (from Burstall [9]) which reverses a list in place.

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

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 $t\ell^0 = \lambda$. 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

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

The following predicates are also needed for the proof:

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

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

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

Unchanged(k) expresses the fact that the tails of each cell in the list refered to by k in σ are not changed from their original values. Distinct(j,k) says that there is no 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 & $i(\sigma) = t\ell\ k(\sigma)$

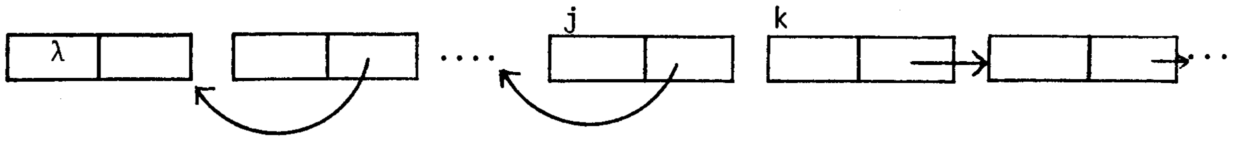
Q_4: Unchanged(i) & Noncircular(i) & Distinct(k,i) & Reverse(k,k) & $t\ell(k(\sigma))(\sigma_0) = i(\sigma)$

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 <u>originally</u> 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 $Q^{\sigma}_{\sigma'}$.

1. Note that Q_0 can be expressed in terms of σ_0:

$$\text{Noncircular}(k): \ \forall p \in TL^{+} \cdot pk(\sigma_0) \neq k(\sigma_0).$$

Q_1 is Unchanged(k) & Noncircular(k) & $j(\sigma) = nil$.

Thus we get

$$\begin{aligned} Q_1(\sigma'): \ & \forall p \in TL^{*} \ \ p(k(\sigma'))(\sigma_0) = pk(\sigma') \\ & \& \ \forall p \in TL^{+} \ \ pk(\sigma') \neq k(\sigma') \\ & \& \ j(\sigma') = nil \end{aligned}$$

Now by axiom 6, $j(\sigma') = nil$, $k(\sigma') = k(\sigma_0)$ & $pk(\sigma') = pk(\sigma_0)$. Hence

$$\begin{aligned} Q_1(\sigma') \equiv \ & \forall p \in TL^{*} \ \ pk(\sigma_0) = pk(\sigma_0) \\ & \& \ \forall p \in TL^{+} \ \ pk(\sigma_0) \neq k(\sigma_0) \\ & \& \ j(\sigma_0) = nil \end{aligned}$$

which clearly reduces to Noncircular(k).

2. We must show that Q_1 & $k(\sigma) \neq nil \Rightarrow Q_2$. Distinct(j,k) holds vacuously, and Reverse(k,j) holds vacuously with $m = 0$. Also, $j(\sigma) = nil$, so $j(\sigma') \neq nil \Rightarrow t\ell(j(\sigma))(\sigma_0) = k(\sigma)$ holds vacuously.

3. $\quad Q_2 \rightarrow Q_3(\sigma')$.

But $\quad Q_3(\sigma') = Q_2(\sigma')$ & $i(\sigma') = t\ell \ k(\sigma')$

Applying axiom 4 and since i does not appear in Q_2, we have $Q_2(\sigma') = Q_2$, while $i(\sigma') = t\ell \ k(\sigma)$ and $t\ell \ k(\sigma') = t\ell \ k(\sigma)$. Hence $Q_3(\sigma') = Q_2$ as required.

4. We must show that

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

where $\sigma' = \leftarrow(t\ell\ k, j, \sigma)$.

(i) Unchanged(i). We must show that

$$Q_3 \Rightarrow \forall p \in TL^* \cdot p(i(\sigma'))(\sigma_0) = pi(\sigma')$$

From Q_3, $\forall p \in TL^* \cdot p(k(\sigma))(\sigma_0) = pk(\sigma)\ \&\ i(\sigma) = t\ell\ k(\sigma)$

$\Rightarrow \quad \forall p \in TL^* \cdot p(t\ell\ k(\sigma))(\sigma_0) = pt\ell\ k(\sigma)$

since $t\ell(k(\sigma))(\sigma_0) = t\ell\ k(\sigma)$

$\Rightarrow \quad \forall p \in TL^* \cdot p(i(\sigma))(\sigma_0) = pi(\sigma)$ since $i(\sigma) = t\ell\ k(\sigma)$

Now from Q_3, $\forall p \in TL^+\ \ pk(\sigma) \neq k(\sigma)$

$\Rightarrow \quad \forall p \in TL^*\ \ pt\ell\ k(\sigma) \neq k(\sigma)$

$\Rightarrow \quad \forall p \in TL^*\ \ pi(\sigma) \neq k(\sigma)$ (*)

Hence $\forall p \in TL^* \cdot pi(\sigma') = pi(\sigma)$ axiom 9 (**)

Thus

$\forall p \in TL^* \cdot p(i(\sigma'))(\sigma_0) = pi(\sigma')$ from axiom 5 and the above derivation.

Hence Unchanged(i).

(ii) Noncircular(i). From (*) we have

$\forall p \in TL^* \cdot pi(\sigma) \neq k(\sigma)$

$\Rightarrow \forall p \in TL^+ \cdot pi(\sigma) \neq t\ell\ k(\sigma)$ applying $t\ell$ to both sides

$\Rightarrow \forall p \in TL^+ \cdot pi(\sigma') \neq i(\sigma')$ since $i(\sigma) = t\ell\ k(\sigma)$ and by axiom 5 and (**).

Hence Noncircular(i).

(iii) Distinct(k,i). We must show that

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

From Q_3, $\forall p_1, p_2 \in TL^* \cdot p_1 j(\sigma) \neq p_2 k(\sigma)$ & $t\ell\ k(\sigma) = i(\sigma)$

$\therefore \forall p_1, p_2 \in TL^* \cdot p_1 j(\sigma) \neq p_2 i(\sigma)$

$\Rightarrow \forall p_1, p_2 \in TL^* \cdot p_1 j(\sigma) \neq p_2 i(\sigma')$ by (**)

But $\forall p \in TL^*$, $pt\ell\ k(\sigma') = pj(\sigma)$ by axiom 9 and Distinct(j,k)

$\Rightarrow \forall p_1, p_2 \in TL^* \cdot p_1 t\ell\ k(\sigma') \neq p_2 i(\sigma')$

Now Noncircular(k) gives us

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

Hence in particular $pt\ell\ k(\sigma) \neq k(\sigma)$

$\Rightarrow pi(\sigma) \neq k(\sigma)$ since $t\ell\ k(\sigma) = i(\sigma)$

$\Rightarrow pi(\sigma') \neq k(\sigma')$ by(**) and axiom 9.

Thus $\forall p_1, p_2 \in TL^* \cdot p_1 k(\sigma') \neq p_2 i(\sigma')$ which is Distinct(k,i).

(iv) Reverse(k,k). We must show that

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

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 Distinct(j,k)$^{(***)}$

Now from Q_3, Reverse(j,k) is

$$\exists m \forall n < m \cdot t\ell^{m-n-1}j(\sigma) = t\ell^{n}k(\sigma_0) \qquad (****)$$

$$=> \exists m \forall n < m \cdot t\ell^{m-n-1}(t\ell\ k(\sigma'))(\sigma') = t\ell^{n}k(\sigma_0) \quad \text{by } (***)$$

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

Also, $t\ell(j(\sigma))(\sigma_0) = k(\sigma)$ from Q_3 if $j(\sigma) \neq$ nil

$=> t\ell(t\ell^{m-1}k(\sigma_0))(\sigma_0) = k(\sigma)$ from (****)

$=> t\ell^{m}k(\sigma_0) = k(\sigma')$ notation and axiom 5.

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

This is Reverse(k,k). If $j(\sigma) =$ nil, then the result holds vacuously with $m = 0$.

v) Finally, it must be shown that

$$Q_3 => t\ell(k(\sigma'))(\sigma_0) = i(\sigma')$$

Now $t\ell(k(\sigma))(\sigma_0) = t\ell\ k(\sigma)$ from Unchanged(k)

$=> t\ell(k(\sigma'))(\sigma_0) = t\ell\ k(\sigma)$ by axiom 5

$=> t\ell(k(\sigma'))(\sigma_0) = i(\sigma')$ since $i(\sigma) = t\ell\ k(\sigma)$ and $i(\sigma) = i(\sigma')$.

5. We must show that $Q_4 \Rightarrow Q_4(\sigma') \;\&\; j(\sigma') = k(\sigma')$.
where $\sigma' = \leftarrow(j,k,\sigma)$.
Note that j does not occur in Q_4, so $Q_4 \Rightarrow Q_4(\sigma')$ by axiom 6 successively applied. Furthermore

$$j(\sigma') = k(\sigma) \quad \text{by axiom 6.}$$

and $$k(\sigma') = k(\sigma) \quad \text{by axiom 6.}$$

6. We must show that $Q_5 \Rightarrow Q_2(\sigma')$
where $\sigma' = \leftarrow(k,i,\sigma)$.

That is, that Unchanged(i) & Noncircular(i) & Distinct(k,i)
& Reverse(k,k) & $j(\sigma) \neq t\ell(k(\sigma))(\sigma_0) = i(\sigma)$
& $j(\sigma) = k(\sigma) \Rightarrow Q_2(\sigma')$.

Now Unchanged(i) $\Rightarrow \forall p \in TL^* \cdot p(k(\sigma'))(\sigma_0) = pk(\sigma')$ by axiom 6.
Noncircular(i) $\Rightarrow \forall p \in TL^+ \cdot pk(\sigma') \neq k(\sigma')$ by axiom 6.
Distinct(k,i) $\Rightarrow \forall p_1, p_2 \in TL^+ \cdot p_1 j(\sigma') \neq p_2 k(\sigma')$
since $k(\sigma) = j(\sigma) \Rightarrow k(\sigma) = j(\sigma')$
and by axiom 6.
Reverse(k,k) $\Rightarrow$ Reverse(k,j) by similar reasoning
Finally $t\ell(k(\sigma))(\sigma_0) = i(\sigma)$
$\Rightarrow t\ell(j(\sigma'))(\sigma_0) = k(\sigma')$ by the same reasoning.

7. We must show that Q_2 & $k = nil \Rightarrow Q_7$

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

The first part follows from Reverse(k,j).

The second part follows since

$$\begin{aligned} & t\ell(j(\sigma))(\sigma_0) = k(\sigma) \\ \Rightarrow\ & t\ell(j(\sigma))(\sigma_0) = nil \quad \text{since} \quad k(\sigma) = nil. \\ \Rightarrow\ & t\ell(t\ell^{m-1} k(\sigma_0))(\sigma_0) = nil \quad \text{since} \quad \text{Reverse}(k,j) \\ \Rightarrow\ & t\ell^m k(\sigma_0) = nil. \qquad \Box \end{aligned}$$

This proof is fairly tedious, since we have been very thorough. When some confidence in the use of the axioms has been gained the proof could be shortened. The development of a notational aid such as the one developed by Reynolds for Arrays (Reynolds [36]) would also help shorten the proof. Nevertheless, we believe that the abstract nature of the model used has made it fairly straightforward to reason about programs using linked lists and that these arguments are direct. We have illustrated the ease with which the proof rule can be applied.

3 Reasoning with the Continuous Model

In the continuous model, a program is viewed as a mapping

$p: L \to L$

taking one structure to another. We can thus write the same in place reverse program as a recursive program, namely

$$F(f) = \lambda e \quad \underline{\text{if}} \ \ x(e) = \text{null} \ \underline{\text{then}} \ \ e \ \ \underline{\text{else}} \ \ f(\hat{R}(e))$$

where $\hat{R}$ is the derived operation

$$R(e) = R(x,i,R(j,x,R(t\ell\ x,j,R(i,t\ell\ x,e))))$$

(We use an informal λ-notation to denote the argument list structure to a function).

Now recall that L_Σ is the canonical Σ-term algebra for lists. The elements of L_Σ maybe thought of as finite or infinite lists. Then suppose that we have the two functions

$$\text{append}: L_\Sigma \times L_\Sigma \to L_\Sigma$$
$$\text{reverse}: L_\Sigma \to L_\Sigma$$

which are the usual functions of LISP, except that $\text{append}(t_1,t_2) = t_1$ if t_1 is infinite. That is

$$\text{append}(x,y) = \underline{\text{if}} \ \ x = \text{null} \ \underline{\text{then}} \ \ y \ \ \underline{\text{else}} \ \ \text{cons}(\text{hd}\ x, \text{append}(t\ell\ x,y))$$

```
reverse(x) = if  x = null  then  x
                    else  append(reverse(tℓ x),hd x)
```

We assume that x is either null or is a list terminating in null (or is an infinite "linear" list).

The function f defines a function

$$|f|_j : L \rightarrow L_\Sigma$$

which is the solution of x in $f(L)$ (i.e. $|x|_{f(L)}$).

The partial correctness of f is obtained by showing that

$$\phi: \quad |f|_j \leq \lambda e \cdot \text{append}(\text{reverse}(|x|_e),|j|_e) \qquad (*)$$

That is, the solution of j in f(e) must be the reverse of the solution of x in e, appended to the solution of j in e. (This is slightly more general than we actually require: the intent of the program is to reverse a given list x, and return to solution as j. If j is null in e, then we get the required assertion. The more general form is needed for the correctness proof).

Any induction method could be used to establish the above relationship. We use the stepwise computational induction of de Bakker and Scott, which is described in detail, for example, in Manna [30]. Briefly, the method may be described as follows:

If $F \Leftarrow \lambda x \cdot \tau(F)$ is a recursive program and $\phi(F)$ is an admissable predicate on F, then if

(i) $\phi(\Omega)$ is true where Ω is the undefined function

and (ii) $\forall f[\phi(f) \Rightarrow \phi(\tau(f))]$

then $\phi(f_p)$ holds, where f_p is the least fixed point of the recursive program F.

In the particular case of (*) it is easily verified that (*) is admissable. Let append(t_1,t_2) be denoted $t_1 \cdot t_2$. We establish that $\phi(f_p)$ is true as follows:

(i) $\phi(\Omega)$ is trivially true

(ii) Suppose that for any f, $\phi(f)$ is true. We must show that $\phi(\tau(f))$ is true. That is, that

$$|F(f)|_j \leq \lambda e \cdot \text{append}(\text{reverse}(|x|_e, |j|_e) .$$

Now $F(f) = \lambda e \cdot \underline{\text{if}}\ x(e) = \text{null}\ \underline{\text{then}}\ e$
$\underline{\text{else}}\ f(\hat{R}(e))$.

Hence

$$\begin{aligned}|F(f)|_j &= \lambda e \cdot \underline{\text{if}}\ x(e) = \text{null}\ \underline{\text{then}}\ |j|_e \\ &\qquad \underline{\text{else}}\ |f(\hat{R}(e))|_j \\ &\leq \lambda e \cdot \underline{\text{if}}\ x(e) = \text{null}\ \underline{\text{then}}\ |j|_e \\ &\qquad \underline{\text{else}}\ \text{append}(\text{reverse}(|x|_{\hat{R}(e)}), |j|_{\hat{R}(e)})\end{aligned}$$

by induction.

We claim that this is $\text{append}(\text{reverse}(|x|_e),|j|_e)$ since

(i) If $x(e) = \text{null}$, then

$$\text{append}(\text{reverse}(|x|_e),|j|_e) = \text{append}(\text{null},|j|_e)$$
$$= |j|_e \quad \text{as required.}$$

(ii) If $x(e) \neq \text{null}$, say e is

$$x = a \cdot t_1 \qquad (\text{cons}(a,t_1))$$
$$j = t_2$$

where $t_1,t_2 \in CT_\Sigma(x)$, then

$$\text{append}(\text{reverse}(|x|_e),|j|_e) = \text{reverse}(a\cdot|t_1|_e)\cdot|j|_e$$
$$= \text{reverse}(a\cdot|t_1|_e)\cdot|t_2|_e$$
$$= \text{reverse}(|t_1|_e)\cdot a\cdot|t_2|_e$$

Now suppose that t_1 and t_2 do not contain any variables. Then $\text{reverse}(|t_1|_e) \cdot a \cdot |t_2|_e = \text{reverse}(t_1) \cdot a \cdot t_2$.
Also, if $x(e) \neq \text{null}$, and if $e' = \hat{R}(e)$, then e' is given by

$$x = t_1$$
$$j = a \cdot t_2$$

by a simple "simulation" of $\hat{R}$. Then

$$\text{append}(\text{reverse}(|x|_{e'}),|j|_{e'})$$
$$= \text{reverse}(|t_1|_{e'}) \cdot |j|_{e'}$$
$$= \text{reverse}(t_1) \cdot a \cdot t_2 \qquad \text{as required.}$$

The condition that t_1, t_2 contain no variables may be viewed as the "input" condition. In fact a weaker condition could have been used, namely that

$$|t_1|_e = |t_1|_{e'}$$

and $$|t_2|_e = |t_2|_{e'}$$

which would be satisfied for example if the variables i, j and x did not appear in t_1 and t_2.

We have thus shown that

$$|F(f)|_j \leq \lambda e \cdot \text{append}(\text{reverse}(|x|_e), |j|_e)$$

and hence that

$$|f_p|_j \leq \text{append}(\text{reverse}(|x|_e), |j|_e).$$

by induction.

We have in fact only established partial correctness of F. We would also have to establish that F was defined whenever $\lambda L_e \cdot \text{append}(\text{reverse}(|x|_e), |j|_e)$ was to establish equality. There are also different possible choices for the partial order $\leq$ used. The most appropriate choice for the particular function used here is that

$$t_1 \leq t_2$$

iff $t_1 = t_2$ or $t_1 = \perp$. If we use the usual ordering on lists, then we have only established that the list returned by f is "less than or equal to" append(reverse($|x|_e$),$|j|_e$) , and some additional work would be required to establish equality.

Because we have shown that the Ω-algebra LIST is continuous, we can view programs as being continuous mappings

$$f: \mathcal{L} \to \mathcal{L}$$

and hence use any induction technique defined for continuous functions. An interesting topic for further study is the development of such proof techniques for list structures with sharing. In the case of the "in place" reverse program presented above, the proof of correctness using the continuous model was much shorter than the invariant assertion proof using linked lists. To what extent can induction rules be used to prove correctness of more complex programs with sharing? Is it possible to prove equivalence of programs not in the strong sense, but in the sense that for any input structure some particular components of the output have the same solution in $\mathcal{L}_\Sigma$?

4 Relation to Other Work

We have shown how the two semantic models developed in this thesis may be used to prove correctness of programs that manipulate data types with sharing and circularity. We believe that the reference model has the same expressive power as the graph - theoretic model of Oppen and Cook [31] but with a much simplified proof rule, and hence allowing for simpler proofs. The reference model is not restricted to any subclass of linked lists and in fact can be used to describe arbitrary graphs with two outedges. It can also be simply extended to cells with n parts by defining a family of cons functions

$$c_n\colon \ \mathrm{Cell}^n \to \mathrm{Cell}$$

and projection functions

$$\delta_i^n\colon \ \mathrm{Cell} \times \mathrm{Struct} \to \mathrm{Cell}$$

in a way similar for example to Elgot [14]. The purpose of the work on the reference model was not to present a proof method, but rather to describe the abstract semantics of references. These semantics could be used as the basis for proof systems, or could be used directly with the proof rule, as discussed in section 2 of this chapter. The proof system thus obtained can be used for arbitrary programs using the type, including the programs presented by Burstall in [9]. We believe that such proofs will be easier than "ad hoc" proofs based on a more operational semantics, although perhaps not as concise as the proofs in Burstall [9]. However, it seems possible that a general proof system with the same power as Kowaltowski's [25] system, but with simpler notation, could be developed. The proof rule for the type is in some sense standard since it is a generalized back substitution rule, and there may be no need to derive a more complex rule.

The difficulty with all the models based on the Scott-Strachey or graph theoretic models seems to arise because of the non-abstract nature of the underlying model. This situation seems analogous to the difference arising from operational versus denotational models of program semantics. The operational or non-abstract models have the disadvantage of needing non-essential details for proofs. The abstract models are easier to work with because of the absence of these details and because the correct high level concepts have been captured.

Very little work on treating shared data types in a continuous setting has been done. Reynolds in [35] has illustrated the inverse limit approach to defining a domain which may be regarded as containing infinite lists but no attempt has been made in that work to discuss sharing. As far as we are aware, no other author has defined shared structures as a continuous algebra or continuous lattice.

The problem of finding the initial algebra in an equationally defined class of continuous algebras has been investigated by ADJ[3] and Courcelle and Nivat [12]. In [3] ADJ investigate rational algebraic theories rather than the less general continuous algebras and obtain a characterization of the initial theory in the class of all rational theories satisfying a set of equations. This characterization is similar to the quotient characterization described in chapter 3, but is stated in terms of theories and is therefore intuitively further from the characterization of non-continuous data types as a quotient of T_Σ than the result using normal forms given in this thesis. Courcelle and Nivat [12] follow the route of "completions" by completing chains and hence obtaining initiality. The interest is, however, the equivalence of programs rather than the characterization of data types.

5 Future Research

We believe that the reference model can be used as the basis of a language for the verification of much more complex programs than the one presented in chapter 5. For example, it would be interesting to attempt a proof of the marking algorithm (Knuth [24]) or other linked list programs. Possibly a more graphic language for list structures could be developed, similar to Reynold's language for arrays [36]. It was indicated in chapter 3 that the technique for including variables and their bindings in the specification of linked lists could also be used for other types, such as stacks. To what extent would this allow the formation of a proof rule for the types?

The continuous model could be a fruitful starting point for a more general analysis of sharing, for example as in functions of "higher level" (Maibaum and Lucena [29]). The important topic of sharing in parallel processing may also be amenable to treatment with an approach similar to that used for defining continuous types. Some more specific question include the following:

- Is it possible to find necessary and sufficient conditions for a class of algebras satisfying a set of equations to have an initial algebra?
- Is it possible to find a sufficient condition which is weaker than the existence of a normal form function?
- Can the results of characterizing the congruence q_e for a regular set of equations e be generalized for a set of recursive equations (which may not be regular)? Is there anything to be gained by extending the results in this way to functions that may be regarded as non-nullary operators?
- Can the continuous model be used for the non-recursive verification methods such as invariant assertions or intermittent assertions?
- Is it possible to establish correctness of the type LIST, for example by comparing it to a more operational type?

We believe that these and other questions could provide some valuable insight into the verification of programs which use data types; the study of data types with sharing; and the study of continuous data types, such as "control" data types. Goguen [16] has suggested that it is possible to treat procedures and control structures in a way that is uniform with the treatment of data types. Thus we could view a program as being a derived operator over an algebra with a continuous domain, and whose operators are the control structures. The results in chapter 4 could be a starting point for investigation of such "control types".

We believe it would be a worthwhile exercise to extend the characterization theorem (theorem 4.12) to recursive equations (from regular equations), since such an extension might allow control structures such as <u>while</u> to be written as "recursive" axioms.

REFERENCES

[1] ADJ - 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, 1976.

[2] ADJ - Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.G., *Initial Algebra Semantics and Continuous Algebras*, JACM Vol. 24 No. 1., 1977.

[3] ADJ - Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.G., *Rational Algebraic Theories and Fixed-Point Solutions*, 17th Annual Symposium on Foundations of Computer Science, IEEE and SIGACT, Houston, Texas, October 1976.

[4] Ambler, Allen L., Good, Donald I., Browne, James C., Burger, Wilhelm F., Cohen, Richard M., Hoch, Charles G. and Wells, Robert E., *Gypsy: A Language for Specification and Implementation of Verifiable Programs*, Proceedings of an ACM Conference on Language Design for Reliable Software, North Carolina, March 1977.

[5] Ashcroft, E.A., *Private Communication.*

[6] Barron, D.W., *An Introduction to the Study of Programming Languages*, Cambridge Computer Science Texts No. 7, 1977.

[7] Berry, G. and Courcelle, B., *Program Equivalence and Canonical Forms in Stable Discrete Interpretations*, Proc. 3rd Colloquim on Automata, Languages and Programming, Edinburgh, 1976.

[8] Burstall, R.M., *Programming Proving as Hand Simulation with a Little Induction*, Information Processing '74, North Holland, Amsterdam 1974.

[9] Burstall, R.M., *Some Techniques for Proving Correctness of Programs Which Alter Data Structures*, Machine Intelligence 7, Edinburgh.

[10] Church, A., *A Formulation of the Simple Theory of Types*, Journal of Symbolic Logic, Vol. 5 (1940) pp. 56-68.

[11] Cohn, P.M., *Universal Algebra*, Harper and Row, 1965.

[12] Courcelle, Bruno and Nivat, Maurice, *Algebraic Families of Interpretations*, 17th Annual Symposium on Foundations of Computer Science, IEEE and SIGACT, Houston, Texas, October 1976.

[13] Earley, Jay, *Toward an Understanding of Data Structures*, CACM, Vol. 4, No. 10, (1961).

[14] Elgot, Calvin, C., *On the Many Facets of Lists*, IBM Technical Report, RC 6449, June, 1977.

[15] Floyd, R.W., *Assigning Meanings to Programs*, in J.T. Schwartz (ed)., Proceedings of a Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, pp. 19-32, American Mathematical Society, New York.

[16] Goguen, Joseph A., *Abstract Errors for Abstract Data Types*, University of California at Los Angeles, Computer Science Department, Semantics and Theory of Computation Report #6, May, 1977.

[17] Guttag, John V., Horowitz, Ellis and Musser, David R., *Some Extensions to Algebraic Specifications*, Proceedings of an ACM Conference on Language Design for Reliable Software, North Carolina, March 1977.

[18] Guttag, J., *Abstract Data Types and the Development of Software*, Communications of the ACM, Vol. 20, No. 6, (June 1977).

[19] Hoare, C.A.R., *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Vol. 12, No. 10, (October 1969).

[20] Hoare, C.A.R., *Proof of Correctness of Data Representations*, Acta Informatica, Vol. 1, pp. 271-281, (1972).

[21] Hoare, C.A.R., *Recursive Data Structures*, International Journal of Computer and Information Sciences, Vol. 4, No. 2, (1975).

[22] Hoare, C.A.R. and Wirth, N., *An Axiomatic Definition of the Programming Language Pascal*, Acta Informatica, Vol. 2, pp. 335-355, (1973).

[23] Kieburtz, Richard B., *Programming Without Pointer Variables*, ACM Sigplan Notices, Vol. 8, No. 2, Proceedings of a Conference on Data: Abstraction, Definition and Structure, Salt Lake City, March 1976.

[24] Knuth, Donald E., *The Art of Computer Programming*, Vols. 1,2 and 3, Addison-Wesley (1973).

[25] Kowaltowski, T., *Data Structures and Correctness of Programs*, Tech. Report, Dept. de Mathemática Aplicada, Universidade de São Paulo, 1976.

[26] Lehmann, Daniel J. and Smyth, Michael B., *Data Types*, Proceedings 18th IEEE Symposium on Foundations of Computing, Providence R.I., Nov. 1977.

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

[28] Liskov, Barbara, Synder, Alan, Atkinson, Russell and Schaffert, Craig, *Abstraction Mechanisms in CLU*, Proceedings of an ACM Conference on Language Design for Reliable Software, North Carolina, March 1977.

[29] Maibaum, T.S.E. and Lucena, Carlos J., *High Order Data Types*, International Journal of Computer and Information Sciences (to appear).

[30] Manna, Zohar, *Mathematical Theory of Computation*, McGraw Hill, 1964.

[31] Manna, Zohar and Waldinger, Richard, *Is "Sometime" Sometimes Better than "Always"? Intermittent Assertions in Proving Program Correctness*, Communications of the ACM, Vol. 21, No. 2, Feb. 1978.

[32] McCarthy, John, Abrahams, Paul W., Edwards, Daniel J., Hart, Timothy P. and Levin, Michael I., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass., 1962.

[33] 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.

[34] Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G. and London, R.L., *Notes on the Design of EUCLID*, Proceedings of an ACM Conference on Language Design for Reliable Software, North Carolina, March 1977.

[35] Reynolds, J.C., *Notes on a Lattice-Theoretic Approach to the Theory of Computation*, Course Notes, Syracuse University, October 1972.

[36] Reynolds, J.C., *Reasoning About Arrays*, (To Appear).

[37] Scott, D.S., *The Lattice of Flow Diagrams*, Lecture Notes in Mathematics, No. 188, Symposium on Semantics of Algorithmic Languages, Springer-Verlag, 1971.

[38] 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.

[39] Standish, Thomas A., *Data Structures: An Axiomatic Approach*, BBN Report No. 2639, August, 1973.

[40] Tompa, F.W., *A Practical Example of the Specification of Abstract Data Types*, Tech. Report, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (1978) 36 pp.

[41] Wulf, William A, London, Ralph L. and Shaw, Mary, *An Introduction to the Construction and Verification of Alphard Programs*, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976.