HIGHER ORDER DATA TYPES

T.S.E. Maibaum*

Research Report CS-77-24

Department of Computer Science
University of Waterloo
Waterloo, Ontário, Canada

Carlos J. Lucena**
Departamento de Informática
PUC/RJ
Rio de Janeiro, RJ, Brasil

September 1977

* This work was partially supported by a research grant from the National Research Council of Canada.

** This work was partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq; Brasil).

HIGHER ORDER DATA TYPES

T.S.E. Maibaum*
Department of Computer Science
University of Waterloo
Waterloo, Ontário, Canada

Carlos J.Lucena**
Departamento de Informática
PUC/RJ
Rio de Janeiro, RJ, Brasil

ABSTRACT

We consider a generalisation of the concept of abstract data type suitable for modelling situations in which there is more than one level of functionality. An instance of such a situation is the difference in level of functionality between the query and update functions in a data base. We introduce the concept of a higher order data type to model this idea. The underlying algebraic ideas are outlined and sample applications of the concept presented.

* This work was partially supported by a research grant from the National Research Council of Canada.

** This work was partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq; Brasil).

1. INTRODUCTION

Most of the current work in programming methodology and langua - ge design (notably for languages of the Algol family) recognizes the convenience of separating the algorithmic and data aspects of a program (e. g., the title of Wirth's book: "Programs = Algorithms + Data Structures"). This separation brought to bear several interesting perspectives for people working in the area of program synthesis. This is because it allows for the independent synthesis of a correct program schema and a correct data representation which together establish the meaning of a program. (It is difficult to synthesize an Algol-like program because it implicitly manipulates data structures). The recognition of the need for the referred separation is analogous to the underlying idea which gave birth to an important area of applications programming, namely: data base system. (In fact, modelling of data base systems was a main motivation in the pursuit of this work).

Some effort have been made to capture mathematically the semantics of programs in which this separation between algorithm and data is precisely characterized ([BUR], [ADJ2], [GUT]). This work has served as a basis for the treatment of the problem of correctness of data representation. That is, starting with an abstract program we can move to more and more concrete representations of its basic type until we get to a suitable implementation of the type (that is, an operational program).

We claim that the algebraic approach to program semantics can further influence the current research efforts in programming methodology and language design. In fact, the algebraic approach can precisely put into perspective the role of what we call higher order data types and their utilization in programming and in the modelling of programming concepts. In fact, programs which accept as parameters variables of type *type* and their proof aspects can be characterized precisely through the algebraic approach. Programs that manipulate higher-order data types can be used to specify an operational model of a data base system.

We should point out immediately that we do not consider such "multi-level" data types as "sequence of stacks" or "queue of sequences" or "stack of stacks" as being examples of higher order data types in our sense of the meaning of higher order. Higher order for us means more than one functional level: we have operations which manipulate data (integers, characters, etc.), operations which manipulate the operations which manipulate data, etc.

Perhaps this concept can be explained more successfully with respect to one for our motivating examples: models for data bases. In thinking about data base systems, we make a clear logical distinction between query functions and update functions in a data base. Query programs on a data base differ from normal programs on "normal machines" in the important respect that the result of exactly the same query may be different before and after an update. (Clearly, if a query asked for the highest salary paid to an employee of the company, the result of the query is max dollars before the update changes max to max+y). Forgetting these update capabilities for the moment, the basic query functions (i.e., the ones built into the so-called query language) together with the valid sets of data on which the query functions could be defined form a system similar to the basic machine functions on a computer (addition, multiplication, tests, etc.) together with the valid sets of data on which these basic machine operations are defined. That is, the query portion of a data base is essentially a special purpose machine with basic query functions as basic machine functions.

Since an update can change the meaning of a query, it must obviously modify the special purpose query machine in some way. This change is usually accomplished by changing the value of a basic query function at some point in its domain from "undefined" or "error" to some defined value or vice-versa. Thus updates can be viewed as operations which manipulate basic query functions (by changing the definition of the latter in some way).

The (multi-level) data type "sequence of stacks" is not an example of a higher order data type since the application of a sequence

operation (such as car, cdr, cons, etc.) to a sequence of stacks does not change the definition of the stack operations (such as top, pop, push). Thus "sequence of stacks" is more akin to the usual notion of machines than to higher order data types. To summarise, higher order data types have the property that the definition of some of the basic operations defined for the type vary with time and that this variation is accomplished by means of some basic "update" operations.

In the sequel, we intend to provide a mathematical development of the theory underlying the use of such data types. In section 2.1 we introduce some basic algebraic concepts. In section 2.2 we define higher order data types and in section 2.3 we outline a method for proving the correctness of implementation of such data types. In section 2.4 we overcome a shortcoming of the previous development - a need to take into account possible non-termination of procedures implementing operations of the type. In section 3.1 we outline the language MADCAP which is particularly appropriate for illustrating the use of higher order data types. Section 3.2 is devoted to a sketch of a program implementing the query and update portions of a data base. The use of higher order data types as a tool in software engineering is outlined in section 3.3 in the development of a line justifier program as motivated in [GRI]. Part 4 then summarises the paper and outlines possible developments.

(We assume a basic knowledge of set theory and some knowledge of algebra. The usual mathematical notation will be followed, including A^* for the set of all finite strings on the alphabet A and λ for the empty string in A).

2. ON THE CONCEPT OF HIGHER ORDER DATA TYPES

The concept of higher order data type is essentially an extension of the algebraic approach to the specification of abstract data types. Thus we will begin our development by quickly outlining the necessary algebraic concepts together with some illustrative examples. (See also [ADJ2], [GUT]). The concept of higher order data type is then developed in terms of an important motivating example of its use: mathematical models for data bases. This is followed by a section on methods of verification for higher order data types. We then go on to consider extensions of the theory necessitated by the fact the implementations of (abstract) data types are programmed and thus have the more general properties of any program (e.g., the possibility that the procedure implementing the operation of the type will not terminate).

2.1. As mentioned in the previous section, an abstract data type is just a collection of (families of) objects together with operations defined over these (families of) objects. We begin by formalising this concept.

Let S be a set of sorts. A many-sorted alphabet sorted by S is an indexed family of sets $\Sigma = \{\Sigma_{w,s}\}_{<w,s>\in S^* \times S}$. An operator (or operation name) $f \in \Sigma_{w,s}$ is said to be of type $<w,s>$, arity w, rank $\ell(w)$ (where $\ell(w)$ is the length of w), and sort s. A (many-sorted) Σ-algebra A_Σ is an indexed family of sets $A = \{A_s\}_{s\in S}$ together with an indexed family of assignments $\alpha = \{\alpha_{w,s}\}$ for $\alpha_{w,s}: \Sigma_{w,s} \to (A_{s_1} \times \ldots \times A_{s_n} \to A_s)$ of operations to operation names (where $w = s_1 \ldots s_n$ and for sets B and C, $(B \to C)$ is the set of functions from B to C). Note that we have dropped the indication of the indexing set ($<w,s> \in S^* \times S$), from $\{\alpha_{w,s}\}$. We will often do this when the indexing set is obvious from the context.

We denote $\alpha_{w,s}(f)$ by f_A for $f \in \Sigma_{w,s}$. We call A the family of carriers of A_Σ and we call A_s the carrier of sort s of A_Σ.

Example 2.1.1:

Let $S = \{N,S,B\}$. Let $\Sigma_{\lambda,S} = \{\Lambda\}, \Sigma_{\lambda,B} = \{T,F\}$, $\Sigma_{S,N} = \{TOP\}$, $\Sigma_{S,S} = \{POP\}$, $\Sigma_{S,B} = \{EMPTY\}$, and $\Sigma_{NS,S} = \{PUSH\}$. For all other $\langle w,s\rangle \in S^* \times S$, $\Sigma_{w,s} = \emptyset$. Consider $A = \{A_N, A_S, A_B\}$ where $A_N = \{0,1,\ldots\}$, $A_S = A_N^*$, $A_B = \{\underline{true}, \underline{false}\}$. Now define the operations assigned to the names in Σ a follows: $\Lambda_A = \lambda$ (the empty string in A_N^*), $T_A = \underline{true}$, $F_A = false$, and for $v \in A_N^* = A_S$ and $n \in A_N$ we have

$$TOP_A(nw) = n,$$

$$POP_A(nw) = w,$$

$$EMPTY_A(w) = \underline{if}\ \ w = \lambda\ \underline{then}\ \underline{true}\ \underline{else}\ \underline{false},$$

$$PUSH_A(n,w) = nw.$$

We have now defined a stack of integers with the obvious operations assigned to the names (see [ADJ2]).

Let A_Σ, B_Σ be Σ-algebras. A Σ-homomorphism $h: A_\Sigma \to B_\Sigma$ is an indexed family of mappings $h_s: A_s \to B_s$ such that for $f \in \Sigma_{w,s}$ and $\langle a_1,\ldots,a_n\rangle \in A^w = A_{s_1} \times \ldots \times A_{s_n}$).

$$h_s(f_A(a_1,\ldots,a_n)) = f_B(h_{s_1}(a_1),\ldots,h_{s_n}(a_n)).$$

So homomorphisms are mappings which "commute" with the application of operations in algebras. Σ-homomorphisms which are injective, surjective and bijective are called Σ-monomorphisms, Σ-epimorphisms, and Σ-isomorphisms, respectively. The composition of two Σ-homomorphisms is a Σ-homomorphism and composition of Σ-homomorphisms is an associative operation.

An algebra I_Σ is said to be initial in the class of algebras A if and only if $I_\Sigma \in A$ and for each $A_\Sigma \in A$ there exists a unique Σ-homomorphism $h_A: I_\Sigma \to A_\Sigma$.

Example 2.1.2:

Given Σ, define the family $T_\Sigma = \{T_{\Sigma,s}\}_{s \in S}$ by:

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

(ii) For $f \in \sum_{w,s}$ and $\langle t_1,\ldots,t_n \rangle \in T_\Sigma^w$,

$$ft_1 \ldots t_n \in T_{\Sigma,s}.$$

T can be made into a Σ-algebra by defining, for $f \in \Sigma_{w,s}$,

$$f_{T_\Sigma}(t_1,\ldots,t_n) = ft_1 \ldots t_n.$$

It is well known that T_Σ is initial in the class of all Σ-algebras. □

An abstract data type is a Σ-algebra for some appropriate Σ. Note that we do not require an abstract data type to be an initial algebra (see [ADJ2]). This is because not all abstract data types are initial algebras as is demonstrated by the following.

Example 2.1.2:

Let $S = \{\bar{n},N\}$. Let $\Sigma_{\bar{n},N} = \{\text{array}\}$ and $\Sigma_{w,s} = \emptyset$ for all other $\langle w,s \rangle$. Let $A_{\bar{n}} = \{1,\ldots,n\}$ and $A_N = \{0,1,2,\ldots\}$. Define $\text{array}_A: A_{\bar{n}} \to A_N$ by $\text{array}_A(i) = 0$ for $i = 1,\ldots,n$. Thus A_Σ is an array of zeros (see also [HO1]). Note, however, that A_Σ is not an initial algebra as defined above for any reasonable class containing it. □

A data structure is an element of the carrier of sort s (for some $s \in S$) of an abstract data type A_Σ. Thus in example 2.1.1, A_Σ is an abstract data type and "012340" is an example of a data structure. (We could extend the definition of data structure in a straightforwared manner by allowing a combination of elements from the carrier of A_Σ).

2.2. We now proceed to discuss the concept of higher order data types. Consider the motivating example of a data base. The basic query functions have the property that the same function name may have different operations assigned to it before and after an update. So we must consider some way of mirroring this change.

Let Σ sorted by S be names for basic query functions and Ω sorted by S' be names for update functions. At any given moment in time, the data together with the current query operations forms some Σ-algebra A_Σ. After an update we get another "query algebra" B_Σ. So we must characterise this collection of different query algebras in some way.

A category $\underline{\underline{C}}$ is a collection of objects $|\underline{\underline{C}}|$ and a family of sets of morphisms $\underline{\underline{C}}(A,B)$ for each $\langle A,B\rangle \in |\underline{\underline{C}}| \times |\underline{\underline{C}}|$ such that:

(i) For each $A \in |\underline{\underline{C}}|$, there is a distinguished morphism 1_A in $\underline{\underline{C}}(A,A)$ called the identity morphism;

(ii) For each $f \in \underline{\underline{C}}(A,B)$, $g \in \underline{\underline{C}}(B,D)$, there exists a morphism $gof \in \underline{\underline{C}}(A,D)$ called the composition of f and g.

Moreover, these classes of morphisms have the property that:

(iii) For $f \in \underline{\underline{C}}(A,B)$, $1_B \circ f = f \circ 1_A$;

(iv) For $f \in \underline{\underline{C}}(A,B)$, $g \in \underline{\underline{C}}(B,D)$, and $h \in \underline{\underline{C}}(D,E)$ we have $(hog)of = ho(gof)$. i.e., composition of morphisms is associative.

We have introduced categories because our group of "query machines" will form a category.

Let $\underline{Alg}_\Sigma$ denote the class of all Σ-algebras with all Σ-homomorphisms between them.

Lemma 2.2.1: $\underline{Alg}_\Sigma$ is a category

Proof: The objects are of course Σ-algebras and the morphisms are Σ-homomorphisms. Identity homomorphisms exist and Σ-homomorphisms are closed under composition. It is also well known that Σ-homomorphisms satisfy (iii) and (iv) above. □

However, $\underline{Alg}_\Sigma$ is too large a category to really model our collection of query machines. This obviously because not every Σ-algebra can be a "query machine" in a given data base since a given data base is tied to some particular representation of the data. So let R be some representation of the families of data being considered. (So $R = \{R_s\}_{s \in S}$). Let $\underline{Alg}_\Sigma(R)$ be the class of Σ-algebras with carrier R and all Σ-homomorphisms between them.

Lemma 2.2.2: $\underline{Alg}_\Sigma(R)$ is a category. □

In fact, $\underline{Alg}_\Sigma(R)$ is a subcategory of $\underline{Alg}_\Sigma$ in the sense that objects and morphisms of the former are also objects and morphisms, respectively, of the latter. Moreover, composition in $\underline{Alg}_\Sigma(R)$ is just a restriction of composition in $\underline{Alg}_\Sigma$.

$\underline{Alg}_\Sigma(R)$ is still too large a category for our collection of query machines because we usually require that the operations assigned to the names in Σ have certain properties. For example, it is clear that in a stack, as defined in example 2.1.1, $TOP_A(PUSH_A(n,w)) = n$. We would require that any definition or implementation of a stack have this property. We can impose this kind of requirement on our collection of query machines as is shown in the sequel.

A Σ-equation of sort $s \in S$ is a pair $\langle t_1, t_2 \rangle \in T_\Sigma(X_w)_s$ (for some $s \in S$). We usually write $t_1 = t_2$ rather than $\langle t_1, t_2 \rangle$. We say that A_Σ satisfies $t_1 = t_2$ if and only if for all assignments $a: X_w \to A$, $\bar{a}(t_1) = \bar{a}(t_2)$. In other words, evaluating t_1 and t_2 after consistently substituting values for variables in both gives the same result in all cases. Now let $E = \{E_s\}$ be a family of sets of equations. A_Σ satisfies E if it satisfies each equation in E separately.

Let $\underline{Alg}_{\Sigma,E}$ be the class of all Σ-algebras which satisfy E together with all Σ-homomorphisms between them. (Classically, $\underline{Alg}_{\Sigma,E}$ is an example of a variety (see [COH])). Let $\underline{Alg}_{\Sigma,E}(R)$ be the class of all Σ-algebras with carrier R and all Σ-homomorphisms between them. Clearly, we have the following result.

Lemma 2.2.3: $Alg_{\Sigma,E}$ and $Alg_{\Sigma,E}(R)$ are categories. (In fact, they are subcategories of $\underline{Alg}_\Sigma$ and $\underline{Alg}_\Sigma(R)$, respectively). □

At this juncture it is important to point out that R represents all objects which could be manipulated by the "query functions" of the data base. The physical addition or deletion of data in the data base is modelled by resetting the result of a query function to some defined value in the former case and some undefined or error value in the latter.

Let us now consider updates. Updates are made using some basic update operations. So let Ω be an $\bar{S}$ sorted alphabet which will be used as names for basic update operations. The operations corresponding to the names in Ω will manipulate operations assigned to the names in Σ, among other things. Thus, in changing the operation assigned to a symbol in Σ, an update causes a mapping of some $A_\Sigma \in \underline{Alg}_{\Sigma,E}(R)$ into some $B_\Sigma \in \underline{Alg}_{\Sigma,E}(R)$. For simplicity, we will assume that updates manipulate the Σ-algebras in $Alg_{\Sigma,E}(R)$ rather than the operations assigned to symbols in Σ. Thus the updates define an algebra U_Ω where $U = \{U_s\}_{s \in \bar{S}}$ and $|\underline{Alg}_{\Sigma,E}(R)| = U_s$ for some $s \in \bar{S}$.

Example 2.2.4: Consider the example of arrays as defined in example 2.1.3. Let $\{1,\ldots,n\}$ be represented, for example, by binary numbers and $\{0,1,2,\ldots\}$ by some binary representation. Call this family of representations R. Σ is a two-sorted alphabet defined by: $\Sigma_{\bar{n},N} = \{\text{array}\}$ and for all other $\langle w,s\rangle \in \{\bar{n},N\}^* \times \{\bar{n},N\}$, $\Sigma_{w,s} = \emptyset$. Thus $\underline{Alg}_\Sigma(R)$ corresponds to the class of all one dimensional (non-negative) integer arrays with n entries. (Note that we have not required any additional properties of the functions named by "array"). Now consider an update to one such array, say $A_1 \in \underline{Alg}_\Sigma(R)$. In a program, such an update is usually specified by some assignment statement such as

$$A_1[i] := e$$

for some non-negative integer valued expression e. The meaning of this statement is that the i'th entry in A_1 is replaced by the value of e. That is, we have changed the function array_{A_1} into the function array_{A_2} defined by

$$\text{array}_{A_2}(j) = \begin{cases} \text{array}_{A_1}(j) & i \neq j \\ e & i = j \end{cases}$$

To model this algebraically, let Ω be a $\{\bar{n},N,r\}$ - sorted alphabet defined by $\Omega_{r N \bar{n},r} = \{\text{assign}\}$ and for all other $\langle w,s\rangle \in \{\bar{n},N,r\}^* \times \{\bar{n},N,r\}, \Omega_{w,s} = \emptyset$.

Let $B = \{B_{\bar{n}}, B_N, B_r\}$ where $B_{\bar{n}} = \{1,\ldots,n\}$, $B_N = \{0,1,2,\ldots\}$ and $B_r = \underline{Alg}_\Sigma(R)$. Define assign_B by

$$\text{assign}_B(A_1, i, e) = A_2$$

where $A_1 \in \underline{Alg}_\Sigma(R)$, $i \in \{1,\ldots,n\}$, e has a value in $\{0,1,2,\ldots\}$ and A_2 is defined to be the algebra in $\underline{Alg}_\Sigma(R)$ which has the operation array_{A_2} as defined above.

We shall not go into how the algebra B can be specified axiomatically but it is clear that, among other properties, we should have at least

$$assign_B(x,i,array_x(i)) = x.$$

It is not immediately clear how such axioms could be used to define (if possible) initiality properties of the algebra B_Ω. This is one of the open problems not treated in this paper. □

Having modelled our motivating example and considered a very simple and different example above, we are now ready to formally define higher order data types. We actually define only order two data types and leave to the reader the generalisation of the concept to higher orders. Let Σ be an S-sorted alphabet, Ω and $\bar{S}$-sorted alphabet, and $R = \{R_s\}$ some family of representations for the data of concern. An <u>abstract data type of order two</u> (or more considely, <u>order two type</u>) is a pair $\langle \underline{Alg}_{\Sigma,E}(R), B_\Omega \rangle$ where:

(i) E is some axiomatic specification of the properties of the operations assigned to the names in Σ;

(ii) $\underline{Alg}_{\Sigma,E}(R)$ is the category of Σ-algebras with carriers R, satisfying E and all Σ-homomorphisms between them (as defined above);

(iii) B_Ω is an Ω-algebra with $B_s = |\underline{Alg}_{\Sigma,E}(R)|$ for some $s \in \bar{S}$.

An <u>order two (data) structure</u> is an element (or combination of elements) in the carrier of an order two type.

2.3. Proof Properties of Higher Order Types

As with the previously studied order one types, one of the reasons for mathematically modelling these higher order data types is to provide ways of studying properties of the types. One of the most important properties is the correctness of an implementation with respect to some abstract definition of the type. For example, suppose that the stack algebra described in example 2.1.1 is implemented by using binary representation for natural numbers, linked lists for stack states, 0 and 1 for true and false, respectively, and appropriately programmed functions to correspond to TOP, POP, etc. This implementation defines an "implementation algebra" M_Σ which,if it is a correct implementation, is isomorphic to A_Σ. (See [ADJ2] for a detailed outline of this concept of correctness).

We want to generalise the methods of proof of correctness of an implementation as described above to higher order types. Let $Y_1 = \langle \underline{Alg}_{\Sigma,E}(R), B_\Omega \rangle$ be an order two type defined in terms of some abstract representation (say, set theoretic) of the data. Let $Y_2 = \langle A(Q), C_\Omega \rangle$ be some purported implementation of the above type. Y_1 and Y_2 should somehow be the "same" if Y_2 correctly implements Y_1. In the case of B_Ω and C_Ω, the requirement is again obviously isomorphism as Ω-algebras. To define sameness for $\underline{Alg}_{\Sigma,E}(R)$ and $A(Q)$ we need the following definitions.

Let $\underline{\underline{A}}$ and $\underline{\underline{B}}$ be categories. A functor $F:\underline{\underline{A}} \to \underline{\underline{B}}$ is a mapping defined in terms of a mapping $|F|:|\underline{\underline{A}}| \to |\underline{\underline{B}}|$, called the object part, and a family of mappings $\{F_{C,D}:\underline{\underline{A}}(C,D) \to \underline{\underline{B}}(|F|(C),|F|(D))\}_{\langle C,D\rangle \in |\underline{\underline{A}}|x|\underline{\underline{A}}|}$, called the morphism part, such that

(i) $F(1_C) = 1_{F(C)}$ for all $C \in |\underline{\underline{A}}|$

(ii) $F(gof) = F(g)oF(f)$ for $f \in \underline{\underline{A}}(C,D)$, $g \in A(D,E)$

(Note that we have dropped the absolute value signs from around $|F|$ and the subscripts from $F_{C,D}$ for the sake of readability). So functors are mappings between categories wich preserve identities and composition. Two categories

$\underline{\underline{A}}$ and $\underline{\underline{B}}$ are isomorphic if and only if there is a functor $F: \underline{\underline{A}} \to \underline{\underline{B}}$ and a functor $G: \underline{\underline{B}} \to \underline{\underline{A}}$ such that $GoF = 1_{\underline{\underline{A}}}$ and $FoG = 1_{\underline{\underline{B}}}$ (where composition of functors is defined in the obvious way in terms of composition of the object and morphism parts of the functors) for the identity functors 1_A and 1_B defined on $\underline{\underline{A}}$ and $\underline{\underline{B}}$, respectively. (The reader may already have asked himself why we demanded that the first component of a two-level type be a category rather than just a collection of algebras with a common carrier satisfying a common system of axioms. The reason shoud now be obvious: we were anticipating the result below. Checking the isomorphism of two categories is a much better structured problem than checking the "sameness" of two collections of algebras).

It should now be clear that Y_1 and Y_2 are the "same" if and only if B_Ω and C_Ω are isomorphic as Ω-algebras and $\underline{Alg}_{\Sigma,E}(R)$ and $A(Q)$ are isomorphic as categories.

We will not belabour the point here, but many of the techniques described in [ADJ2] for order one types, can be generalised to higher order types in a straight-forward manner.

2.4. Programmed Higher Order Data Types

We now come to a discussion of a serious shortcoming common to this treatment so far and many previous discussions of abstract data types. The main purpose of a mathematical definition of abstract data types is to check the correctness of some implementation of the data type. However, any implementation is via programs used to define the operations involved. As such, these implementation types are subject to the usual problems of programs in general. For example, a call to a procedure implementing some operation may not terminate. This problem is partly foreseen in [ADJ2] where the problem of defining what the definition of $TOP_A(\lambda)$ should be is discussed. (Note that λ is the empty stack whereas TOP_A must give some result in $\{0,1,\ldots\}$). To introduce "error" elements into the carriers of the algebras involved (as advocated in [ADJ3]) is not enough because non-termination is not a detectable error condition. (This is not to say that

"error" elements may not play a useful role for explicating other pheno - mena). So to correct this fault, we must use the methods used by semanticists to treat the problem of non-termination of programs. Luckily, although the underlying mathematics is more complicated, the proof rules remain generally the same.

We do not propose to motivate the use of the following mathematical treatment for the study of the semantics of programming languages. (For this we refer the reader to [ADJ1], [LEV]). We quickly present some definitions and appropriate extensions of material in the previous sections.

We proceed to introduce into our domains elements denoting non-terminating computations and undefined results (i.e., the "result" of a non-terminating computation). We will use these new elements to partially order our domains. For example, we can say that the completely undefined function $\perp$ "aproximates" any function in the sense that where they are both defined, they agree. Also, the program segment

if b then (S;loop) else null

(where null is a statement meaning do nothing and loop puts the program in a non-terminating loop) approximates in the same sense as above the program segment

while b do S.

Let us formalise these ideas. A partially ordered set (or poset) is a pair $(D, \sqsubseteq)$ where D is a set and $\sqsubseteq$ is a reflexive, antisymmetric, and transitive binary relation on D. We usually write just D for (D,E). $X \subseteq D$ is said to be a countable chain (or ω-chain) if X is totally ordered. We usually write the ω-chain $X = \{x_0, x_1, \ldots\}$ as $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ or as $\{x_i\}_{i \in N}$. A poset D is an ω-complete partially ordered set (or cpo) if and only if D has a least element denoted by $\perp$ (called bottom) and every ω-chain x_i in D has a least upper bound. The least upper bound (or lub) of a set $X \subseteq D$, denoted by $\sqcup X$, is an element $x \in D$ such that for all $y \in X$, $y \sqsubseteq x$ and if there is some $z \in D$ so that for all $y \in X$, $y \sqsubseteq z$, then $x \sqsubseteq z$.

Let D and E be partially ordered sets. $f:D \to E$ is monotonic if and only if for $x \sqsubseteq y$ in D, $f(x) \sqsubseteq f(y)$ in E. f is said to be ω-continuous if and only if $f(\sqcup X) = \sqcup f(X)$ for all ω-chains $X \sqsubseteq D$ whenever $\sqcup X$ exists in D. Let Σ be a many-sorted alphabet. A Σ-algebra A_Σ is said to be an ω-continuous ω-algebra if and only if each A_s is a cpo and each f_A (for $f \in \Sigma_{w,s}$, some $\langle w,s \rangle$) is continuous as a function from its domain to its codomain. (The domain of f_A is $A_{s_1} \times \ldots \times A_{s_n}$ which is a cross-product of cpo's and so is a cpo. See [ADJ1] for further elaboration). A Σ-homomorphism $h:A_\Sigma \to B_\Sigma$ for continuous Σ-algebras A_Σ and B_Σ is ω-continuous if each h_s is ω-continuous.

We now modify our definition of abstract data type by requiring that the algebra involved be a continuous Σ-algebra for some Σ. We now generalise Example 2.1.2 to obtain an initial algebra in the class of all ω-continuous Σ-algebras together with ω-continuous Σ-homomorphisms between them. (We now drop ω from ω-continuous and ω-complete).

Let Σ ambiguously denote $\bigcup_{\langle w,s \rangle \in S^* \times S} \Sigma_{w,s}$. Consider the set of all partial functions from some set A to some set B, denoted by $[A \multimap B]$. We can order $[A \multimap B]$ by set inclusion on the graphs of the partial functions (considered as subsets of AxB). It can easily be shown that with this ordering we have a cpo. If $f:A \to B$ is a partial function, let $def(f) = \{a \mid \langle a,b \rangle \in f$. def(f) is the domain of definition of f. For each $s \in S$, let $CT_{\Sigma,s}$ be the set of partial functions $t:N^* \to \Sigma$ such that

(i) If $\lambda \in def(t)$, then $t(\lambda)$ has sort s;

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

(a) $w \in def(t)$;

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

Let $CT_\Sigma = \{CT_{\Sigma,s}\}$ and define f_{CT} for $f \in \Sigma_{w,s}$ by:

(i) If $w = \lambda$ (and so f is a constant or nullary), then $f_{CT_\Sigma} = \{<\lambda, f>\}$;

(ii) If $t_i \in CT_{\Sigma,s_i}$ for $i \leq i \leq n$, then

$$f_{CT_\Sigma}(t_1,\ldots,t_n) = \{<\lambda,f>\} \cup \bigcup_{i<n} \{<iu,g> | <u,g> \in t_{i+1}\}.$$

Thus CT_Σ is a Σ-algebra and is in fact a continuous Σ-algebra. Moreover, if $\underline{CAlg}_\Sigma$ is the class of all continuous Σ-algebras with strict, continuous Σ-homomorphisms between them, then CT_Σ is initial in $\underline{CAlg}_\Sigma$. (A Σ-homomorphism is strict if it maps the least element of the domain onto the least element of the codomain).

We can now define $\underline{CAlg}_\Sigma(R)$ analogously to the definition of $Alg_\Sigma(R)$ for some family R of cpo's and show that $\underline{CAlg}_\Sigma$ and $\underline{CAlg}_\Sigma(R)$ are categories. If we now want to specify properties of continuous Σ-algebras, we can use inequations of the form $t_1 \sqsubseteq t_2$ for $t_1, t_2 \in CT(X_w)_s$. ($CT_\Sigma(X_w)$ is defined analogously to $T_\Sigma(X_w)$.). $t_1 = t_2$ is now a short form for the pair of inequations $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_1$. The concept of a continuous Σ-algebra satisfying an inequation $t_1 \sqsubseteq t_2$ is defined in the same way as before. Denote by $\underline{CAlg}_{\Sigma,E}$ and $\underline{CAlg}_{\Sigma,E}(R)$ the generalisations of $\underline{Alg}_{\Sigma,E}$ and $\underline{Alg}_{\Sigma,E}(R)$ respectively. $\underline{CAlg}_{\Sigma,E}$ and $\underline{CAlg}_{\Sigma,E}(R)$ are clearly categories.

A <u>(generalised) order two type</u> is a pair $<\underline{CAlg}_{\Sigma,E}(R), B_\Omega>$ where:

(i) E is some axiomatic specification using inequations of the properties of the operations assigned to the names in Σ;

(ii) $R = \{R_s\}_{s \in S}$ is family of cpo's and $\underline{CAlg}_{\Sigma,E}(R)$ is the category defined above;

(iii) B_Ω is a continuous Ω-algebra with $B_s = |\underline{CAlg}_{\Sigma,E}(R)|$ for some $s \in \bar{S}$.

There is one point in the above definition which needs some clarification. If B_Ω is a continuous algebra, then each B_s must be a cpo. However, $|\underline{CAlg}_{\Sigma,E}(R)|$ is just an unstructured set of algebras. We now show how we can regard this set as a cpo in a very natural way. Since CT_Σ is initial in the class of all continuous Σ-algebras, for each $A_\Sigma \in |\underline{CAlg}_{\Sigma,E}(R)|$ there exists a unique, strict, continuous Σ-homomorphism $h_A: CT_\Sigma \to A_\Sigma$. Thus we can represent $|\underline{CAlg}_{\Sigma,E}(R)|$ by the set $H = \{h_A: CT_\Sigma \to A_\Sigma \mid A_\Sigma \in |\underline{CAlg}_{\Sigma,E}(R)|\}$. Order H by $h_A \sqsubseteq h_C$ if and only if for all $t \in CT_{\Sigma,s}$ (any $s \in S$), $h_A(t) \sqsubseteq h_C(t)$ in R_s. (Recall that for any $A_\Sigma \in |\underline{CAlg}_{\Sigma,E}(R)|$, $h_A(t) \in R_s$). This is clearly a partial order on H. We must now show that H is a cpo. Let $D_\Sigma \in |\underline{CAlg}_{\Sigma,E}(R)|$ be defined by: For any $f \in \Sigma_{w,s}$, $f_D(a_1,\ldots,a_n) = \perp_s \in R_s$. That is, all operations in D_Σ are completely undefined. Clearly, h_D is the least element in H. Now let $\{h^{(i)}\}$ be a chain in H. We will define an algebra A_Σ so that $h_A = \bigsqcup h_{A(i)}$ for $h^{(i)}_\Sigma : CT_\Sigma \to A^{(i)}_\Sigma$.

For $f \in \Sigma_{w,s}$, define f_A by

$$f_A(a_1,\ldots,a_n) = \bigsqcup f_{A(i)}(a_1,\ldots,a_n).$$

It is obvious that $\{f_{A(i)}(a_1,\ldots,a_n)\}$ is a chain in R and so f_A is well defined. We must show that A_Σ as defined is a continuous Σ-algebra; i.e., that each f_A is continuous. For this, it is enough to show that

$$f_A(a_1,\ldots,\bigsqcup_{j\in N} a^{(j)},\ldots,a_n) = \bigsqcup_{j\in N} f_A(a_1,\ldots,a^{(j)},\ldots,a_n)$$

for some chain $\{a^{(j)}\}$ in R_{s_k}, $1 \le k \le n$.

$$\begin{aligned} f_A(a_1,\ldots,\bigsqcup_{j\in N} a^{(j)},\ldots,a_n) &= \bigsqcup_{i\in N} f_{A(i)}(a_1,\ldots,\bigsqcup_{j\in N} a^{(j)},\ldots,a_n) \\ &= \bigsqcup_{i\in N}(\bigsqcup_{j\in N} f_{A(i)}(a_1,\ldots,a^{(j)},\ldots,a_n) \\ &\quad \text{since each } f_{A(i)} \text{ is continuous} \\ &= \bigsqcup_{j\in N}(\bigsqcup_{i\in N} f_{A(i)}(a_1,\ldots,a^{(j)},\ldots,a_n) \\ &= \bigsqcup_{j\in N} f_A(a_1,\ldots,a^{(j)},\ldots,a_n). \end{aligned}$$

It should be clear that h_A is an upper bound for $\{h^{(i)}\}$. It is not hard to prove that it is the least upper bound. Thus H is a cpo and by the one to one correspondence between H and $|\underline{CAlg}_{\Sigma,E}(R)|$, so is $|\underline{CAlg}_{\Sigma,E}(R)|$. Thus there is no inconsistency in saying that B is a continuous Σ-algebra.

The proof methods outlined in the previous section have clear analogies in this new setting.

3. ON THE USE AND IMPLEMENTATION OF HIGHER ORDER DATA TYPES

In the following sections we will present two examples which use higher order data types. The first of these examples in section 3.2 is an implementation of the query and update portions of a data base with respect to some given representation of the data. In section 3.3 we illustrate the use of higher order data types in the synthesis of programs by writing a linejustifier program for a text editor.

Both these examples make use of the novel features of an extended version of the language MADCAP which is presented in section 3.1.

3.1. The Language MADCAP-S

MADCAP-S [MOR, MKN] is a block structured and very high level language which has been designed to facilitate program structuring and verification. It is a very high level language in the sense that it uses general sets and sequences (of the kind introduced by the SET-L language [SCH] as data types. It facilitates the structuring and verification of programs by using a special version of the cluster [LIZ] mechanism and by disallowing side-effects in its various features (thus enabling a Hoarelike [HOZ] axiomatization of the language semantic).

In MADCAP-S the concepts of function and procedure are similar to the ones used in EULER and ALGOL-68. A function definition describes the input parameters (if any), the computation to be performed and value to be returned. A function is a value belonging to the type FUNCTION and as such can be assigned to a variable or shared as part of a structure. In general, a function has the following form:

$$\ll [P;] \left\{ \begin{matrix} \rightarrow f \\ S[;\rightarrow f] \end{matrix} \right\} \gg$$

where P stands for the parameter part which is a list of explicitly typed variables, S stands for a non-empty list of statements and f stands for an expression, which is the value to be returned by the function. The following

is an example of a function

<< a in REAL; b in FUNCTION;

x ← b(a); → x^3 + 3a>>

In MADCAP-S a data type is a domain over which values of variables of that type may range. A data type may be either unstructured or structured. The following are unstructured data types: reals, integers, naturals, positive integers, atomic strings, boolean and funtion constants.

The structured types are SET and SEQUENCE. If we call UNIVERSE the set of all values which can be manipulated by a MADCAP-S program, that is ,

UNIVERSE $\stackrel{\text{def}}{=}$ unstructured types ∪ structured types

then

SET ≡ 2^{UNIVERSE} and

SEQUENCE ≡ $\text{UNIVERSE}^{<\infty}$ (≡ UNIVERSE^{*}).

In the definition of structured types and of the type function constant, the following entities can be used: standard data types, data type identifiers and data type expressions.

A data type expression is either a data type identifier or a special expression involving the following operators and other components:

(i) Powerset and binomial (e.g., if T ≡ 2^{INTEGER}, type T is the set of all sets of integers; if T ($\binom{\text{INTEGER}}{3}$), type T is the set of all sets of integers with cardinality 3);

(ii) Cross product and integral exponentiation (e.g., if T ≡ REAL x INTEGER, type T is the set of all pairs formed by a real and an integer; if T ≡ INTEGER^2,type T is the set of all pairs of integers);

(iii) Standard data types (e.g., the use of the type identifiers INTEGER and REAL in the examples i and ii above);

(iv) Data type identifier (e.g., if $A \equiv$ INTEGER x INTEGER is followed by $T \equiv 2^A$, type T is the set of all sets of pairs of integers).

A type function constant is defined by expressing the domain and the range of all functions that belong to the type. That is,

$$T \equiv \langle\langle T_1, T_2, \ldots, T_n \rightarrow T' \rangle\rangle.$$

A variable declaration of the form x in T ← f; establishes the fact that $x \in T$ and assigns to x the value of the expression f.

A function cluster or simply a cluster in MADCAP-S [LUC, LAU] is a special type of external function. The general form of a cluster specification is the following:

```
V  cluster  <<Declaration of formal parameters;
              Declaration of variables and shared variables;
              S;
              F1 ← <<P1;D1;S1; → f1 in T1>>;
              F2 ← <<P2;D2;S2; → f2 in T2>>;
              ..............................
              Fn ← <<Pn;Dn;Sn; → fn in Tn>>;
              →  <F1,F2,...,Fn> or {F1,F2,...,Fn}>>
```

where:

(i) The variable V is of type cluster $\langle\langle FP_1; FP_2, \ldots; \rightarrow FUNCTION^{<\infty} \rangle\rangle$, where the FP_i are its formal parameters. (If the cluster returned a set of functions, that is, $\rightarrow\{F_1, F_2, \ldots, F_n\}$, the type of V would be cluster $\langle\langle FP_1; FP_2; \ldots; 2^{FUNCTION} \rangle\rangle$).

(ii) S is a list of statements acting on variables of the cluster.

(iii) The declaration of shared variables in V is a list of MADCAP-S declarations that specify the types of a set of variables that can only be modified by the functions $F_1,F_2,\ldots,F_n$ (i.e. they are shared by $F_1,F_2,\ldots,F_n$). Note that this is the only situation in a MADCAP-S program in which globals can be assigned values.

(iv) $F_1,F_2,\ldots,F_n$ are variables of the type function constant that contain the specification $(P_i;D_i;S_i;f_i)(1\leq i\leq n)$, of the cluster operations whose effect is defined over the shared variables.

If A1,...,AN are variables of the type function constant, and V is a variable of the type cluster, then the assignment

<A1,...,AN> ← V( );

would make A1 = F_1, A2 = F_2,...,AN = F_n for a cluster V without parameters, (If now A is of type set of functions, the assignment

A ← V( );

would make $A.F_1 = F_1$, $A.F_2 = F_2$, etc.).

Actual referente to the operations encoded in a function cluster are made through the operation do. In the case of the above assignments

do A1; do A2;...; do AN
(and do $A.F_1$; do $A.F_2$;...; do $A.F_n$)

would execute the operations of the cluster. The first operation in a program referring to a cluster also initializes it. The effect of a do is felt only on the shared variables which then retain their values until the next invocation of the cluster.

3.2. A Data Base Implementation

We want to implement a data base in a way which clearly indicates the logical difference between queries and updates (see [MAI1], [MAI2]). We use the concept of cluster since this construct implements abstract data types. There will be two clusters illustrated: a (variable) QUERY cluster and an UPDATE cluster. The QUERY cluster uses some given representation REP of the data that can be stored in the data base. The UPDATE cluster uses REP as well as the QUERY cluster.

a) The body of the UPDATE cluster:

```
cluster  <<REP, QUERY ≡ FUNCTION^<∞;
         w in REP ← ¢ body of REP cluster ¢;
         v in QUERY ← ¢ body of QUERY cluster ¢;
         update1 ← <<¢ the update function uses those access
                       functions of REP which alter the state
                       of REP's data structures ¢
                       ¢ body of update1 ¢
                    → <v1 (w), v2 (w),...,vn(w)> >>
           ⋮
         updaten ← <<   ¢ body of updaten ¢
                    → <v1 (w), v2 (w),...,vn (w)> >>

                 → <update1,...,updaten> >>
```

b) The body of the QUERY cluster:

```
cluster <<
        query1 ← << REP in FUNCTION^<∞ ; z in REP;
                ¢ the query function uses those access
                  functions of REP which do not alter the
                  state of REP's data structures ¢
                ¢ body of query1 ¢ >>
        :
        queryn ← << REP in FUNCTION^<∞ ; z in REP;
                ¢ body of queryn ¢ >>
                        → <query1, ..., queryn> >>
```

The QUERY cluster needs very little explanation. Each operation (of query1, ..., queryn) defined in the cluster takes as argument parameters of type REP where REP is the name of a cluster implementing the representation of the data in question. (Thus REP is a sequence of functions because the REP cluster defines the representation of the data in terms of some simple operations defined on the data. These simple operations are then returned by the cluster REP).

The QUERY cluster returns a sequence of operations <query1, ... , queryn> defined on z of type REP.

The UPDATE cluster takes as arguments (global to the operations defined in the cluster) the REP and QUERY clusters. The operations of the cluster (update1, ..., updaten) are defined in terms of parameters w of type REP and v of type QUERY. Now updatei is defined by altering the state of REP (either by adding more data, removing data, changing existing data, etc.) and redefining the query functions query1, ..., queryn so that they are meaningful over this new state. Note that since v is of type QUERY and query returns a sequence of operations, v_j refers to the j'th element of this sequence (i.e., queryj) and $v_j(w)$ refers to the changed definition of

queryj making it meaningful for w, the new state of REP. Thus updatei returns as a result the sequence of query functions <queryl, ...,queryn> redefined over the new state of REP.

Note that the QUERY and UPDATE clusters are independent of the definition of the REP cluster. A change in the REP cluster need not be reflected by a change in the QUERY and UPDATE clusters (unless the change in REP defines a new type REP' which implements an algebra not isomorphic to the algebra implemented by REP). Similarly, the cluster UPDATE is also independent of the definition of the cluster QUERY (with reservations similar to those above). Thus redefining the basic query functions should not affect the meaningfulness of the update functions. This reflects the idea that changing the access mechanism in a data base should not be "visible" to the user.

3.3. Higher Order Data Types as a Programming Aid

We illustrate the use of higher order data types as an aid in program synthesis by developing a program to justify lines in a text editor. (The problem is described in more detail in |GRI| although the philosophy guiding the synthesis is very different from ours).

A line justifier inserts extra blanks in the spaces between the words on a line so that the last character of the last word on the line appears immediately to the left of the right margin (i.e., in the last column of the line). The number of blanks between pairs of words in a justified line may differ by at most one extra blank. For aesthetic reasons, the insertion of these extra blanks is done alternately on the left and on the right depending on the parity of the line number on the page.

So, supposing we have some representation of a line (on a page); we must write a justifier program which takes this representation an changes it to the representation of the justified line. However, this justifier must be alternatively transformed to insert extra blanks on the left or the right depending on the parity of the line. This immediately suggests a higher order data type approach and our resulting program is as follows:

Line Justifier Example

(i) Line justifier program:

```
<<r, y in (FUNCTION 3); x in (FUNCTION 2); f, g in FUNCTION;
     y ← UPDATE-LINE cluster <<...body of cluster>> (x,r);
     Repeat
     f ← do y.input-line ( );
     if   do f = 1 then g ← do justify-left ( )
                   else g ← do justify-right ( );
     until do g >>;
```

(ii) body of the UPDATE-LINE cluster:

```
UPDATE-LINE cluster << ¢ r stands for the set of access
                       functions of the representation ¢

r in (FUNCTION 3) , m in (FUNCTION 2) ;

input-line  ← << do r.read( ) → {m.parity(r)} >>;

justify-left ← << do r.assign(2, do r.select(2) + do r.
                  select(5) / (do r.select(4) - 1));
                  do r.assign(1, do r.select(2) + 1);
                  do r.assign(3, do mod(do r.select(5),
                  (do r.select(4) - 1)) + 1);
                  do r.assign(5,0);
                            → {m.last-line(r)} >>;

justify-right ← <<do r.assign(1, do r.select(1) + do
                  r.select(5) / (do r.select(4) - 1));
                  do r.assign(2, do r.select(1) + 1);
                  do r.assign(3,mod(do r.select(5),
                  (do r.select(4) - 1)) + 1);
                  do r.assign(5,0);
                            → {m.last-line(r)} >>;
                            → {justify-left, justify-right,
                               input-line} >>;
```

(iii) body of the LINE-INFORMATION cluster:

```
x,m ← LINE-INFORMATION cluster <<
      last-line ← << rep in (FUNCTION 3); boll in BOOLEAN;
                     ¢ checks if current line is the last line
                       in a text ¢
            bool  ←  if do rep.select(6) = MAX
                        then true
                        else false
                              → bool >>;
```

```
parity ← << rep in (FUNCTION 3); p in N;
                ¢ indicates via 1 and 2 the parity of
                the line ¢
           p ← do mod(do rep.select(6),2)
             → {last-line, parity} >>;          → p >>;
```

(iv) body of the REP cluster:

```
r, rep ← REP cluster  << TUPLE ≡ N^6;
        ℓ in TUPLE ← Ω ;
            assign ← << i, j in N;
                        ℓ_i ← j >>;
            select ← << i in N;
                        → ℓ_i >>;
            read   ← << read(ℓ) >>;
                   → {read,assign,select} >>;
```

Let us explain our global strategy first. The main program starts by assigning to the variable y (of the type "set of three functions") the cluster UPDATE-LINE with parameters x and r. x is of type "set of two functions" and will be used to name the operations defined by the LINE-INFORMATION cluster. r is of type "set of three functions" and will be used to name the operations defined by the REP cluster (and thus, indirectly, to refer to the lines of text being justified).

The major portion of the main line program then consists of a repeat ... until ... statement which takes each line in turn and justifies it by inserting blanks in the appropriate way. What is appropriate for a given line will of course depend on the parity of the line. The first statement in the line assigns to f (of type FUNCTION) the result of executing the function named input-line in the set of functions y. (Note that y.input-line takes no parametes since in the cluster UPDATE-LINE the parameters r (representing the operations defined in REP) and m (representing the operations defined in

LINE-INFORMATION) are global to the definition of all the operations). Thus f is assigned the function parity defined on the representation of the current line under consideration (namely r).

The second statement in the body of the repeat ... until ... statement tests the parity of r (by executing f) and, depending on the result, assigns to the FUNCTION variable g the result of executing justify-left or justify-right. Now the execution of either of these functions returns as a result the definition of the function last-line with respect to the representation r of the current line. (This execution also has the important side affect of justifying the line!). Thus the statement do g results in the execution of g (i.e., last-line) on r. Now, the execution of last-line returns a boolean value as a result. This boolean value is true when we have reached the last line of the text. Thus do g acts as the termination condition for our iteration.

Thus two important updates are made. First, the function parity is updated to define the parity of the current line under consideration. This then allows us to choose between the execution of justify left and justify right. Secondly, last-line is updated by the execution of justify-left or justify right so that this function can be used to control the termination of the iteration.

As to the three clusters involved:

(i) The REP cluster implements a representation of lines of text in terms of a 6-tuple of natural numbers (to be described below) and three operations defined on tuples:

(a) The operation assign(i, j) inserts the value j in the i'th position of the 6-tuple. It returns no value as a result.

(b) The operation select(i) returns as a result the value of the i'th component of the 6-tuple.

(c) The operation read just reads a line of text.

The representation ℓ in TUPLE has componentes whose meanings are as follows:

ℓ_1 is the number of blanks before ℓ_3.

ℓ_2 is the number of blanks after ℓ_3.

ℓ_3 is the number of the word after which the pattern of blanks in the line changes.

ℓ_4 is the number of words in the line.

ℓ_5 is the number of blanks after the last word in the line.

ℓ_6 is the number of the line in the text.

Note that ℓ is initialized to "undefined" at the beginning (i.e., before any lines are read).

(ii) The LINE-INFORMATION cluster is to provide just that: critical information about the line under consideration. (This cluster has no global variables or initialization). The operation last-line takes as argument a set of three functions called rep (which is of type REP). Thus do rep.select(6) executes the function named select in rep with the parameter 6 and results in the line number of the current line. This number is then compared to the number of the last line in the text and the result of this comparison is the result of last-line. Similarly, parity returns as a result the line number of the current line (the result of do rep.select(6) modulo 2.

(iii) The UPDATE-LINE cluster has two global parameters (to represent the set of operations defined by REP and LINE-INFORMATION). The operation input-line reads a line of text (by executing do r.read( )) and returns as a result the function named parity defined for the line just read. The operation justify-left performs some calculations necessary to change the representation of the line to the representation of the justified line (to be described below) and returns as a result the operation named last-line defined for the current line under consideration. There are four different calculations made in the body of justify left (using the statements do r.assign(---). Each such statement assigns the result of the calculation to a component of the tuple representing the line. The first statement assigns to the second component the value $(\ell_2+\ell_5)/(\ell_4-\ell_1)$. ($\ell_i$ is the result of do r.select(i)). The calculations made in the other statements are just as easy to describe. Finally, the analysis of the operation justify-right is analogous to that of justify-left.

We leave the analysis of this example in terms of algebras to the reader.

4. CONCLUSIONS

We hope we have illustrated the usefulness of a generalisation of the concept of abstract data type. Higher order types arose from efforts to model an important class of "objects" presently of great interest, namely data bases. The model arose out of a need for a clear distinction between the functional orders (or logical types) of the query and update functions in a data base. The sample program in section 3.3 should also have illustrated how these concepts may be used in the structuring of data (and thereby of programs).

A number of other programming applications of higher order types come to mind. Consider a situation in which some attributes of variables can change over time (in a non-predictable way) with the simple restriction that the set of possible attributes is known ahead of time. This is the case, for example, in SNOBOL where it is not clear whether get next x means the next byte, half-word or word. However, an implementation could be written in which there was only one get function which could be updated dynamically depending on the current attributes of the variable. A similar situation occurs when one writes programs for a network of heterogeneous machines. A ready instruction should be defined in a single cluster and be updated appropriately depending on which machine is being addressed. The portability of programs could also be enhanced by using "updates" in situations similar to the above.

Higher order types can be implemented by language features which are used to model data abstractions. Among these, of course, are the concept of cluster and the availability of types which can take functions as values. An important method of implementing higher order types which has not been mentioned here is the concept of redefining a function by manipulating the body of the function definition. Such a facility is available in LISP, but is usually frowned upon as bad programming practice. (This view might perhaps be explained by the previous lack of methods of viewing the use of such changes in a structured way).

One can view the use of higher order data types as a beneficial structuring concept. Data types can now be viewed as hierarchical structures which may be designed top down. Modularity is enhanced in a number of ways. In the SNOBOL example mentioned above (as well as the related example which followed) the tendency is to overmodularize: write a different module for get for each different situation. The concept of a higher level type prevents this overmodularization. The opposite extreme of undermodularization is made less likely by having a new tool for the structuring of logically different classes of operations.

REFERENCES

[ADJ1] GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G., and WRIGHT, J.B., Initial Algebra Semantics and Continuous Algebras, JACM Vol. 24, No.1, pp. 68-95, 1977.

[ADJ2] GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G., and WRIGHT, J.B., Abstract Data Types as Initial Algebras and Correctness of Data Representations, Conference on Computer Graphics, Pattern Recognition, and DAta Structure, Los Angeles, 1975.

[ADJ3] GOGUEN, J.A. THATCHER, J.W., WAGNER, E.G., and WRIGHT, J.B., Private Communication.

[BUR] BURSTALL, R.M., An Algebraic Description of Programs with Assertions Verification and Simulation, Proceedings of an ACM Conference on Proving Assertions About Programs, New Mexico, 1972.

[COH] COHN, P.M., Universal Algebra, Hamperthow, 1965,

[GRI] GRIES, D., "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs", IEEE Transactions on Software Engineering, Vol. 2, Nº 4, 1976.

[GUT] GUTTAG, J.V., The Specification and Application to Programming of Abstract Data Types, Ph.D. Dissertation, University of Toronto, 1975.

[HO1] HOARE, C.A.R., Notes on Data Structuring, in Structured Programming by O.J.Dahl, E.W. Dykstra, and C.A.R. Hoare, Academic Press, 1972.

[HO2] HOARE, C.A.R., An Axiomatic Basis for Computer Programming, Communications of the ACM, 12:10, 1969.

[LUC] LUCENA, C. et alii, On the Use of Very High Level Languages for the Design and Implementation of Reliable Programs, RJ 1584, IBM Research, 1975.

[LAU] LAUTERBACH, C.H. et alii, Abstract Data Types in MADCAP-VI, Technical Report, Computer Science Department, University of California, Los Angeles, 1975.

[LEV] LEVY, M.R., Doctoral dissertation (in preparation), University of Waterloo.

[LIZ] LISKOV, B. and ZILLES, S. Programming with Abstract Data Types - Proceedings of the SIGPLAN Symposium on Very High Level Languages, 1974.

[MAI1] MAIBAUM, T.S.E., Mathematical Semantics and a Model for Data Bases, Proceedings of IFIP Congress'77, Toronto.

[MAI2] MAIBAUM, T.S.E., Data Bases, Data Types and Semantics, in preparation.

[MOR] MORRIS, J.B. et al, Progress Report on the Language MADCAP-6, Los Alamos Laboratory, University of California, Jan. 1974.

[MKN] MELKANOFF, M.A. et alii, Implementation of MADCAP6 on UCLA's 360/91 IM LAC SYSTEM, UCLA-ENG-7438, School of Engineering and Applied Science, University of California, Los Angeles, 1974.

[SCH] SCHWARTZ, J.T. Abstract Algorithms and a Set-Theoretic Language for Their Expression, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1971.

[WIR] WIRTH, N., Algorithms + Data Structures = Programs, Prentice Hall 1976.

[ZIL] ZILLES, S.N., Abstract Specifications for Data Types, IBM Research Laboratory, San Jose, California, 1975.