THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

A DEFINITIONAL TECHNIQUE FOR SPECIFICATION AND IMPLEMENTATION OF DATA TYPES

Khosrow Hadavi

CRL-TR-16-83

Under the Direction of
Professor Keki B. Irani

APRIL 1983

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]This research was supported by the Department of the Army, Ballistic Missile Defense Advanced Technology Center, Rome Air Development Center, and the Defense Mapping Agency under contract F30602-80-C-0173, and by the Air Force Office of Scientific Research/AFSC, United States Air Force under AFOSR contract F49620-82-C-0089. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

ABSTRACT

A Definitional Technique for Specification
and Implementation of Data Types

by
Khosrow Hadavi

Chairman: Keki B. Irani

A model is proposed for specification and implementation of data types. This model is based on a novel multi-level graph structure, viz. the Σ-structures. A universal set of data structure operators are defined to characterize the data types. Constructive definitions of these operators are presented to the extent that the task of defining them is reduced to specification of only three first order predicate expressions. Using these predicates, the important issue of error is easily and automatically taken care of. This is achieved through the use of functions whose domains are defined by the above predicates in such a way that every constructor operation results in a "non-error" configuration.

The type manipulation operations (TMO) are introduced in order from define data types of a different behaviour to those of the existing ones. An example of a TMO is the "embed" operation. It enables one to combine two data types so that the resulting data type exhibits a behaviour which can be automatically derived from the operand data types.

This facility may also be used to parameterize data types. Another TMO is the enrichment operation. Two types of enrichment are introduced. These are enrichment for "convenience", and enrichment for "change of behaviour." Sufficient conditions are developed in order to distinguish one from the other.

It is demonstrated that the proposed model is highly "extensible." For example, with a very minor alteration, a **stack** specification may be changed to that of a **queue** and vice versa. Also demonstrated is the ability of the specification to define parallelism of the operations. To this end, without any extra burden on the user, highly parallel operations may be defined for updating and/or accessing data.

It is shown that our specification technique offers an invaluable tool to ensure the "security" of a data base, or an operating system. It is also demonstrated that different views of the same set of data may be held by different users concurrently by assigning different predicates to each user.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF FIGURES viii

LIST OF APPENDICES ix

CHAPTER

I- DATA STRUCTURE ABSTRACTION-PROBLEMS AND MOTIVATIONS 1

II- A NEW MODEL 11

2.1- Description of The Model 11

2.2- Components of the Model 12

III- DATA STRUCTURE OPERATIONS 42

3.1- Characteristic Set of Operations 44

3.2- A Definition of Data Types 62

3.3- Data Types as Lattices 66

3.4- The Primitive Data Types 86

3.4.1- Operations to Characterize Primitive Data Types 89

3.4.2- Constructors of Primitive Data Types 92

3.4.3- Primitive Data Types as Posets . . 95

IV- COMPLETENESS AND TYPE MANIPULATION OPERATIONS 100

4.1- Completeness and Soundness 101

4.1.1- Deduction of Terms 105

4.2- Type Manipulation Operations 111

V- ENRICHMENT, EQUIVALENCE AND ERROR 121

5.1- Enrichment of Data Types 121

5.1.1- Changing Behaviour by Enrichment . 122

5.1.2- Auxiliary Operations 125

5.1.2.1- Equational Theory 127

5.2- Equivalence of Data Types 133

5.3- Implementation by Emulation 140

5.4- Error 146

5.4.1- Domain of DSOs 149

VI- CONCLUSIONS AND FURTHER WORK. 152

BIBLIOGRAPHY 175

LIST OF FIGURES

Figure

2.1- A 3-LEVEL Σ-STRUCTURE, Σ_3 17

2.2- STRUCTURES DESCRIBED BY A PREDICATE . . . 20

2.3- DATA STRUCTURE CONFIGURATIONS 40

3.1- REACHABILITY OF ELEMENTS IN Z 74

4.1- REMOVAL OF INNER STRUCTURES BY /-FUNCTION 118

5.1- STRUCTURAL SIMILARITY 140

5.2- ARRAY IMPLEMENTATION OF BINARY TREE . . . 144

LIST OF APPENDICES

APPENDIX A- EXAMPLES 159

APPENDIX B- DYNAMIC CHANGE OF BEHAVIOR 165

APPENDIX C- λ-NOTATION 167

APPENDIX D- MORE ON INSERTION AND DELETION . . 169

CHAPTER I

DATA STRUCTURE ABSTRACTION-PROBLEMS AND MOTIVATIONS

Three phases may be identified with almost every formal approach to realization of data types. These are the concept, the specification, and finally the implementation. Assuming the concept or the desired object to be specified is already known to us, our task is to identify the two remaining areas, i.e. the specification and representation. More specifically, these two concepts parallel the "behaviour", and the "storage representation" [AFS 80] of data structures. Behaviour of data structures deals with its properties, i.e. the features which are unique to that particular data structure. Storage representation is a real world object, i.e. physically tangible, that meets all the requirements of an abstract specification. Consider a **stack** data structure, its behaviour is characterized by a last-in-first-out discipline. The representation of **stack**, on the other hand, may be a **linear list** or a **linked list** or any other storage representation which exhibits a LIFO characteristic.

Problems regarding the storage representation of data structures, concerning efficiency and optimality, were a

popular research topic, particularly, when the non-numeric computing became widespread and the volume of data started to expand very rapidly because of the industry-wide usage of computers. The large volume of data, either in database environments or in numerical problems triggered a number of studies concerning storage of data, some of these are [DIM 69, HAN 69, PAT 69, RAN 72, ROS 71, WON 75, KAR 75, ROS 78].

Behavioral study of data structures has been the main theme of research for the past decade or so. The main thrust has been towards developing a formal technique to specify the behaviour of data structures. This led to the concept of *abstract data types*. An abstract data type defines a class of abstract objects which is completely characterized by the *operations* available on those objects [LIS 74].

In general there are two main categories in which the specification of abstract data types may fall: operational and definitional[GUT 77]. In the operational approach, instead of describing the expected properties, one provides a methodology by which a desired type may be built. The definitional approach, on the other hand, provides a set of axioms , under which the properties are upheld.

The definitional techniques are more formally oriented; their most valuable merit is claimed to be the ability to describe data structures without giving any bias as to how it should be implemented. This lack of implementation bias,

however, may sometimes prove to be not so desirable when an implementation is sought. Two of the definitional techniques which have won a good deal of support are the axiomatic and the algebraic techniques. Both of these techniques describe the behaviour of data structures by a set of axioms which are defined on the operations of data structures. As a consequence of the abstract descriptions, such techniques are normally difficult to construct, comprehend and verify. The algebraic method [GUT 75, ZIL 74, GOG 75] has shown the most potential for automatic implementation and automatic correctness proofs. This is accomplished through the use of equational theory which is of central importance in this scheme. However some crucial issues such as identification and detection of errors are difficult to handle and implement in the theory. A more in-depth, though still incomplete attempt in this area is due to Goguen et. al. [GOG 78]. Their proposed technique requires additional axioms to be added to handle error instances. The so called error equations make the task of specification, in their own words, "..unbelievably complicated" [GOG 78]. Yet one of the principal reasons to have a formal approach is the ability to express errors. Specification of concurrency and other performance related topics are also issues to be considered and solved.

In general, there are a number of factors to be considered and satisfied in a specification technique. These are briefly described below.

-<u>Constructability and Comprehensibility</u>

In general we are interested in constructing a specification with a reasonable degree of confidence which is comprehensible to humans as well as machine. Some authors maintain the view that comprehensibility is not an important issue, since a formal specification is always comprehensible to a machine. We believe that it is just as important for the human user to comprehend the underlying specification since: a) a comprehensible scheme is of value in construction phase of specification; it enhances the degree of confidence, and b) users may exchange their ideas without any ambiguity and the tedium of line by line description of the conventional programs.

-<u>Minimality</u>

Every specification should be free from any extraneous information. For any specification, we are concerned very little about how it should be done; we are only interested in what function(s) should be performed. In addition, minimizing the specification reduces the number of properties to be verified in correctness proofs.

-<u>Applicability</u>

It is obvious that for any specification technique, a wide range of applicability is desired. In other words, how universal or how limited is the power of a specification technique is in order to describe data structures in general.

-<u>Extensibility</u>

It is desirable that a small extension of a concept results in a similar small modification in its specification. For example, in changing a specification of a **set** to a **bag**, it is undesirable to undertake major changes in the original specification of the **set**.

The techniques which have been developed so far are rich in some aspects and deficient in some others. Graphical techniques have the potential of enhancing certain attributes of a specification, for example constructability and comprehensibility. However one problem remains, and that is the minimality issue. Due to the presence of the extra graph structures, the specification is no longer minimal. The algebraic approach, on the other hand, treats operations as letters of alphabet, T. T^*, the free monoid generated by T, is provided with a finite set of equations between certain elements of T. Every instance of the data type is then given by a word W in T^*.

The specification technique which we have proposed is a synthesis of operational and definitional techniques, combining the ease of use and constructability of the operational methods, and the advantages of formality and applicability exhibited by the axiomatic and algebraic methods. Built into our model are concepts such as ease of construction, parallelism of operations, easy extension and the ability to handle "error." Many of these concepts have

been either left out or lightly considered by the existing methods.

In the proposed model the behaviour of data types is specified by the operations defined on their graph structures. This approach is substantiated by the fact that behaviour and structure are intimately related [BRO 80]. Furthermore, the use of pointers, directed graphs or any other unnecessary implementation directed information has been avoided. Despite the many claims that are made regarding the lack of implementation bias in the algebraic approach, there are numerous common examples where without the use of "hidden functions," the technique would not yield favourable results [MOI 80, MAJ 77A]. As a result extraneous information, i.e. implementation bias, is introduced not only to make the technique convenient, but also feasible. In spite of all this, there are still some simple and commonly used data types, for which it is not easy to employ algebraic technique [GOG 78]. We argue that in order to ease the task of specifications we have to make some trade-offs.

Following the tradition of the recent past we also have adopted the principle that data types are algebras. However graph structures are also introduced to gain in the areas where the implicit methods have demonstrated weaknesses. A number of these areas are: constructability, error treatment, comprehensibility, extensibility, and

performance constraints. Guidelines to build complex structures as well as provision of direction for the implementor are among other problems associated with the implicit methods. We have employed graph structures as an aid to describe "relations" between data items; but unlike Earley's approach we do not adopt different types of nodes. It is also unnecessary, as cited in Shneiderman [SHN 74] and Majester [MAJ 77B], to use directed graphs to impose implementation bias. Among the previous studies employing graphs, only the approach of Majester offers any adequate formalism, the rest are either incomplete or insufficient theories. In Majester's work only the relation between abstract entities and the operations characterizing the data structures are considered. The important notion of error and other crucial concepts such as parameterized types have not been considered adequately.

In our approach we have developed a mechanism by means of which one can build complex structures in terms of the more primitive ones. The appealing feature of this process is that the resulting complex structure retains all the properties of the constituent structures. Consider, for example, the list-of-trees (lot) data type. Once we have defined a **list** and a **tree**, by either the user or the host language, a **lot** data type, as will be shown later, is no more than (say) an infix expression with **list** and **tree** as operands:

$$\mathbf{lot} = \mathbf{list} \times \mathbf{tree}$$

As pointed out earlier **lot**, like any other data type, requires a set of operations to characterize its behaviour. Our intention has been to define the "embed operation" (x) so that the "characteristic set" of operations for **lot** is formed automatically out of the characteristic operation set of the constituent **list** and **tree** data types.

Having defined **lot** as above, one can extend the data-type **lot** to contain parameter types, e.g. list-of-tree-of-strings (**lots**) is formed as follows:

lots = lot x string

A very important criterion, rarely considered is the question of performance. The method which we have adopted, enables the user to specify his intended data type as well as to emphasize, if so desired, some performance criteria concerning both the specification and implementation. Specification of parallel operations, for instance, is an important criteria. None of the previous methods have presented the ability to specify concurrency of operations such as parallel insertion and/or parallel deletion. Performance issues may also be specified at the implementation level. As an example of an implementation related performance, consider a two-dimensional array specified as the "embed" of two arrays:

matrix=array x array

With an appropriate terminology, one can also specify the layout, of the data items, either in a row-major order or in a column-major order.

Use of the directed edges,to serve as the access path, may also prove to be another tool to specify performance criteria. The reader should be reminded that although having the ability to specify some performance measures would introduce information extraneous to a minimal specification, we are merely offering the *choice*. In our approach the edge between two nodes (say) does not necessarily imply an access path (link, pointer etc.) in its corresponding memory representation.

One of the major problems of the algebraic approach is to define a set of operations on the desired data structures in a way that would "completely" characterize its behaviour. In addition, it is also necessary to define the operations so that "error" or illegal results (configuration or instances) are avoided, or at least identified. The error treatment of data types, treated lightly in the literature, is an important and difficult problem as exemplified by Goguen et al [GOG 78]. We believe that the recognition of error instances of data structures using a graphical approach is a much easier task than when an axiomatic technique is employed. For example consider a **stack** data type. The error instances of "empty-stack" and "full-stack" are immediately obvious when a graph structure is

visualized. This same information is not quite as obvious when an algebraic approach is employed, unless some "constructive" description of it is visualized [MAJ 77A].

Using our approach of combining simple data types to get more complex ones, the complex data types' error conditions are easily and automatically met by those of the constituent types put together. This facility is made possible by the nature of the type manipulation operations, which preserve the properties of the constituent data types.

The next chapter will introduce the basic components of the model. It will be followed by chapter 3 where we introduce the data structure operations and formally define the characteristic set of operations. The type manipulation operations are examined in chapter 4. The concept of equivalence is discussed in chapter 5 where we also discuss the notion of error and enrichment. Finally chapter 6 contains the conclusions of this study as well as suggestions for further work.

CHAPTER II

A NEW MODEL

2.1- Description of the Model

Graph structures have proved to be useful visual aids in many areas of science. Their presence is of value to human brain and our better comprehension of the underlying system. To a formal computing system, however, graph structures are no more than sets and relations. Although their presence may or may not be exploited for any "biasing" purposes such as implementation guidelines, we have opted to leave the potentials open-ended. However, there are many instances for which the use of the graph structures imposes some unnecessary and extraneous information on the underlying specification. We are willing to accept this in order to gain in other aspects of a specification scheme discussed in the previous chapter.

The concept of multi-level graph structures is introduced to depict relationships between data objects or any other type of elements in general. Following the minimal description techniques of the algebraic advocates,

the operations defined on these graph structures will determine the characteristic and the behavioural aspect of the structures. It will be seen that certain predicates are necessary to define the correct relationship between data elements. Such predicates are particularly useful since they ensure correctness of operations carried out on data structures. These predicates will be referred to as *p-expressions*. the p-expressions merely represent structure of data and not the behaviour of it. For instance a **list** structure may have the same structure as that of a **stack**, but their behaviours are completely different. The combination of both structure and behaviour is referred to as a "data type."

In the remainder of this chapter some basic components of the model are introduced in order to lay the groundwork for what follows.

2.2- Components of the Model

Definition 2.2 below introduces the concept of structures contained within other structures. Hence the notion of multi-level structures is evolved. We have termed them as Σ-structures due to their close resemblance to many-sorted Σ-algebras. Each level of the structure may be employed to represent one or more "sorts." The distinction is not significant, since two sorts or more may be merged into, and considered as, one sort. In addition the relation between the elements of the algebra is emphasized

by means of the "u-relations." Emphasis on both structure and behaviour has many advantages as we shall see, and also noted by[BRO 80].

The following definitions and concepts are designed to serve as a formal framework for the proposed model.

Definition 2.1

A graph is a pair g=(N,E), where N is a set of nodes, and E is a set of unordered pairs in N, called edges.

Following the BNF notation, in the definition given below, a vertical bar(|) indicates "or," and the symbol "ε" denotes "contains." In this definition, the concept of nodes "containing" other nodes, i.e. identifying different "levels" of nodes is introduced. A collection of elements $\langle x_1, x_2, \ldots, x_n \rangle$ where multiple occurrences of the same element is allowed is referred to as a "bag." Finally (x,y) indicates an ordered pair whereas (x;y) denotes an *unordered* pair. The function MAX, below, acts on a set(of integers), S, such that if MAX(S)=s then $s \varepsilon S$ and for every $r \varepsilon S$, $s \geq r$; also MAX({})=∅.

Definition 2.2

An *L-level Σ-structure* Σ_L, is defined as follows:

$\Sigma_L = (S_{L1}, u_{L1})$

$S_{lk}=\{S_{lki}: i=\emptyset \text{ or } =\emptyset, I_{lk}, I_{lk}+1, \ldots, I_{l,k+1}-1$ AND

$I_{lk}=MAX(\{j: S_{lk'j}\varepsilon \bigsqcup_{q<k} S_{lq}\})+1 \quad \}$

$l=1,2,\ldots,L \text{ \& } k\geq 1$

$S_{lki} \in (S_{l-1,i}, u_{l-1,i}) \qquad l>1 \text{ \& } i\neq\emptyset$

$u_{lk} \subseteq \langle (x,y)^n : x,y \in S_{lk} \text{ \& } n\varepsilon N^+\rangle \qquad l=1,2,\ldots,L$; $k=1,2,\ldots$

$S_{1ki} \in \Sigma_{undefined}$ for all $i\geq 1 \mid \Sigma_\emptyset=(S_{\emptyset i}, u_{\emptyset i})$ for all $i\geq 1$.

$\Sigma_{undefined}$ is an undefined Σ-structure.

$S_{\emptyset k}=S_{\emptyset ki}$ for some $i\varepsilon N$ & $k=1,2,\ldots$

$u_{\emptyset k}=\langle\ \rangle$

$S_{\emptyset ki}$ is a *primitive node*.

S_{lki} is a *Structure Support Node* (SSN) for $l,k,i \geq 1$.

$S_{lk\emptyset}$ is the *empty structure (node)* for $l,k\geq\emptyset$.

N^+ is the set of non-negative integers.

A Σ-structure is said *to terminate with primitive nodes,* if it possesses level zero nodes. Similarly a Σ-structure is said *to terminate with SSN,* if all of its lowest level SSN's support $\Sigma_{undefined}$. Henceforth, unless otherwise specified, all structures terminate with SSN.

The following observations are of interest:

- The primitive nodes do not contribute to the number of levels of the Σ-structure.

• The indices l, k, and i assign a numbering of the nodes as follows:

l- level indicator, sometimes referred to as the *first index*.

k- index of the node of the most recent structure within which the current structure is being defined. This may also be referred to as the *second index*.

i- index number of the current-level nodes, referred to as the *third index*.

Hence each node is distinctly indexed.

• Nodes S_{lki} with $i=\emptyset$ are referred to as *empty structures*. Thus if a node $S_{l+1,k,j}$ contains $S_{lj\emptyset}$, then the latter implies that there exists a defined lth level structure , though no "insertion" has yet been made at this level. In effect $S_{lj\emptyset}$ acts as a "place-holder." Henceforth the existence of these place-holders, where applicable, will be implicit in our notation. Furthermore, an empty node $S_{lk\emptyset}$ implies the existence of other emty nodes at levels l-1, l-2, ... , 1 (or $\emptyset$ if the structure terminates with primitive nodes) contained in $S_{lk\emptyset}$. Thus $S_{lk\emptyset}$ contains $S_{l-1,\emptyset,\emptyset}$, and the latter contains $S_{l-2,\emptyset,\emptyset}$ and so on. Note that there may be more than one $S_{l,\emptyset,\emptyset}$ at a level l. We shall see that this ambiguity will not cause any problems since no "activity" may be associated with the content of an empty node.

In contrast to the empty structure, there may also exist the *undefined structure*. An undefined structure would

only occur at the lowest level, such that if level 1 contains the undefined structure then no structure may, or can, be present at any of the level 1 nodes. The distinction between the undefined and empty structures may be shown by the following analogous situation in a high level language. When an array_of_integers is declared, we have a structure which is defined but empty until it is initialized. However, when an array structure is defined without any parameters, the structure which may be contained in each node of the array is undefined.

- If the structure terminates with primitive nodes, a level 1 node, indexed k, may contain any integer, i, labeled <Ø,k,i>.

- A *u-relation* is defined as a bag of unordered pairs of nodes. Thus each u_{lk} is a u-relation.

We shall employ the notation "i<•>j" to designate (i,j) is an element of a u-relation. That means: i *is u-related to* j and j *is u-related to* i. Since it is understood that both (i,j) and (j,i) are in a u-relation, we only need to consider the bag of pairs where only one of the two pairs is present. To be consistent in our approach we only consider the subset, u, of every u-relation such that if $(i,j) \varepsilon u$, then $j \geq i$. Furthermore, for our immediate goals we shall treat the u-relations as sets rather than bags since the bag concept is not of much importance in specifying the "behaviour" of data types. However for "implementation"

purposes, namely the actual memory representation of data types, the bag may prove to be a valuable tool.

Example

A 3-level Σ-structure. The desired structure is depicted below in a graphical manner. The nodes are labeled (lki) to denote S_{lki}.

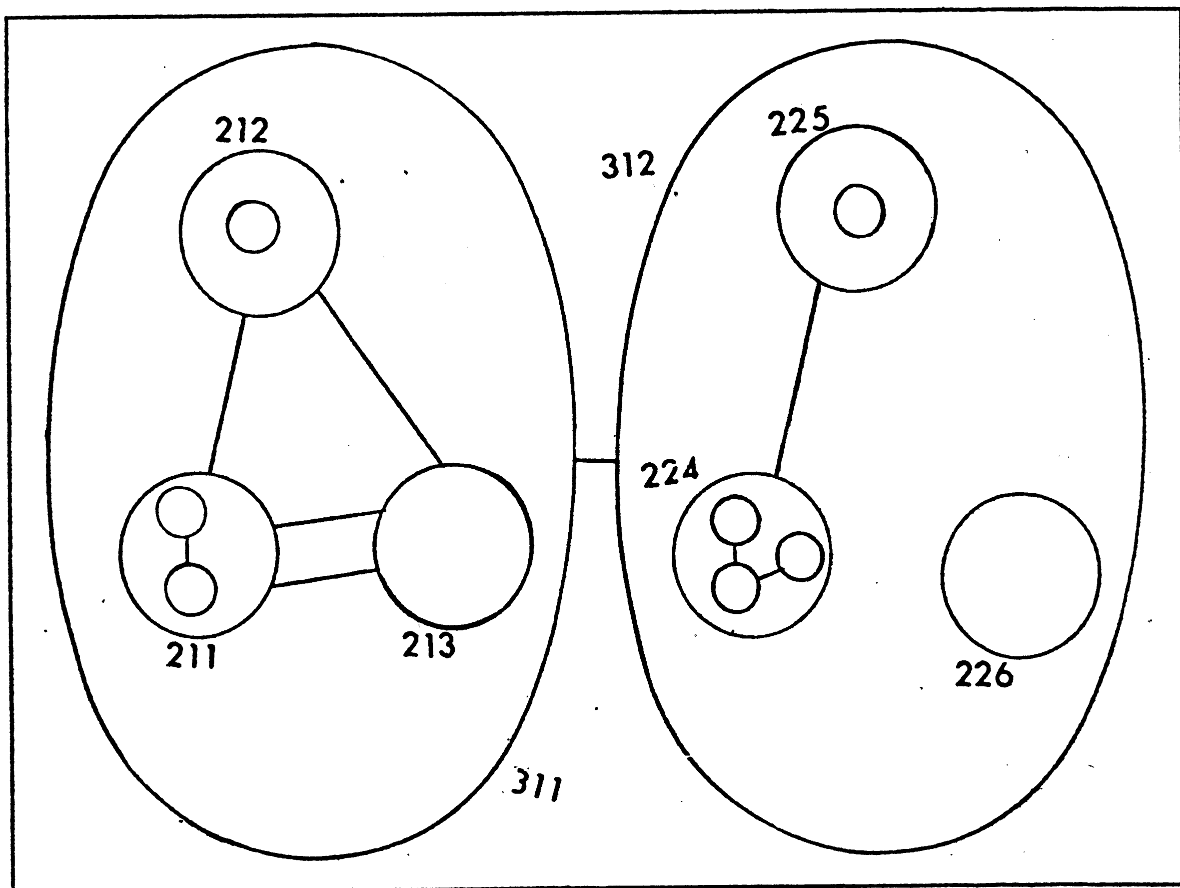


Figure 2.1
A 3-level Σ-structure, Σ_3.

Let us now derive the description for the above structure using the definition of Σ-structures given earlier. Note that S_{lkm} denotes a node $S_{l,k,m}$.

level 3

$\Sigma_3=(\{S_{311},S_{312}\}, <(S_{311},S_{312})>)$

level 2

$S_{311}\in(\{S_{211},S_{212},S_{213}\}, u_{21})$

$u_{21}=<(S_{211},S_{212}),(S_{212},S_{213}),(S_{211},S_{213}),(S_{211},S_{213})>$

$S_{312}\in(\{S_{224},S_{225},S_{226}\},<(S_{224},S_{225})>)$

level 1

$S_{211}\in(\{S_{111},S_{112}\}, u_{11})$ $u_{11}=<(S_{111},S_{112})>$

$S_{212}\in(\{S_{123}\}, u_{12})$ $u_{12}=<\ >$

$S_{213}\in(\{S_{13\emptyset}\}, u_{13})$ $u_{13}=<\ >$

$S_{224}\in(\{S_{144},S_{145},S_{146}\}, u_{14})$

$u_{14}=<(S_{144},S_{145}),(S_{144},S_{146})>$

$S_{225}\in(\{S_{157}\}, u_{15})$ $u_{15}=<\ >$

$S_{226}\in(\{S_{16\emptyset}\}, u_{16})$ $u_{16}=<\ >$

Finally each S_{1ki}, for all k and i, contains $\Sigma_{undefined}$, hence the structure terminates with SSN.

Let u_l denote the union of u_{lk} and S_l be the union of S_{lk} over all values of k. For notational convenience we may denote Σ_L by a pair (S,u) where S is the set of all nodes in Σ_L and u is the union of u_l for all levels l.

Definition 2.3

The *content* of a node S_{lkj}, $C(S_{lkj})$, is defined to be the l-1 level Σ-structure contained in S_{lkj}.

As we shall see shortly, in general we are interested in dealing with a set, not necessarily finite, of Σ-structures collectively. Such a set would contain elements of the form $\Sigma=(S,u)$. In order to be able to describe the desired set of Σ-structures we need to impose a number of restrictions. In the presence of these restrictions we can employ certain predicate expressions, known as "p-expressions", to describe the desired set of Σ-structures.

Example

In this example we present the basic idea of how we may use a predicate, such as $<\cdot>$ which means "is u-related to", in order to describe list-like structures. Later on we shall show that a p-expression would perform the same task but for a whole class of structures. Let i denote S_{lki}. Also let us assume that the domain of values of i is: $\{1,2,3,...,I-2\}$. Then,

a) The predicate expression to describe a list structure, Σ_1, of length $(I+1)$ is: $(\forall i)(i<\cdot>(i+1))$.

This is illustrated in figure 2.2(a).

b) A list where each node is linked to the two nodes immediately following it may be represented by:

$$(\forall i)((i<\cdot>(i+1))_{U}(i<\cdot>(i+2))$$

Cite figure 2.2(b).

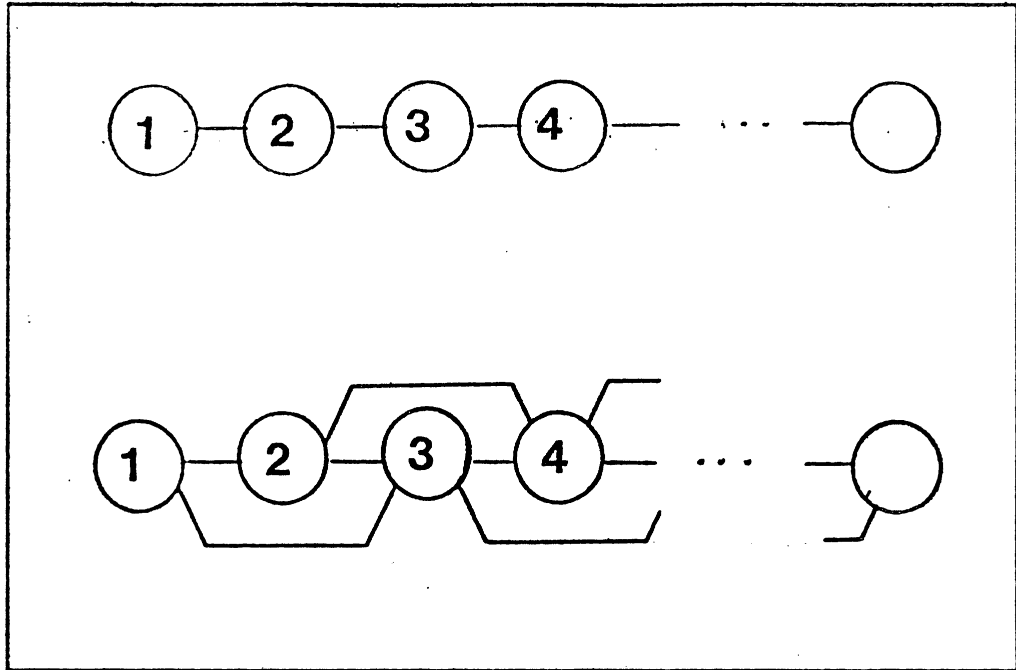


Figure 2.2
Structures described by a predicate.

Notation:

t_q^l- A traversal function (to be defined later) for level l indexed by q.

Σ_L^r- A variable symbol to denote a L-level Σ-structure in SIG_L.

S_l^r- A variable symbol to denote a set of all nodes at level l of Σ_L^r.

S_{lk}^r- A variable symbol to denote a set of nodes at level l of Σ_L^r, contained in the kth node of level l+1.

S^r- $\bigsqcup_l S_l^r$

u_l^r- A variable symbol to denote a u-relation at level l of Σ_L^r.

u^r- $\bigsqcup_l u_l^r$

- A node $S_{l,k,m}$ may be abbreviated to <l,k,m>; or if l and k are clear from the context the index m may be used to represent $S_{l,k,m}$.
- The symbols i,j,lki,lkj (and a few others which would be clear from the context if used) will be employed as variable symbols to denote elements of S_{lk}^r.
- Note that symbols with a superscript r denote variable symbols of the corresponding set. That is Σ_L^r denotes any element of the set SIG. Thus instead of using the phrase "for all Σ_L^r" we may use "for all r."

Consider now a set of Σ-structures:

$$SIG=\{\Sigma^r \;:\; \Sigma^r=(S^r,u^r)\},$$

we may write $u=\{u^r \;:\; \Sigma^r \varepsilon SIG\}$; i.e. u is the set of u-relations each of which is associated with an element of SIG. Our intention is to associate a family of p-expressions: $p_1,\ldots,p_L$ with a set of L-level Σ-structures SIG such that u_l^r, the u-relation at level l of Σ_L^r, may be described by p_l, $0 \le l \le L$, for all elements of SIG. Before any further elaboration on the nature of the p-expressions let us state our first restriction on the set of Σ-structures, SIG.

i) Every element $\Sigma_L^r \varepsilon SIG$ has the same number of levels L and all Σ_L^r terminate with the same level, either SSN or primitive nodes.

Thus we may refer to a set of L-level Σ-structures as SIG_L.

Definition 2.4

A partial computable function, $t : S_{1k} \rightarrow S_{1k}$, such that whenever $t(S_{1,k,i})=S_{1,k,j}$ then $j \geq i$ is referred to as a *traversal function*.

Since a traversal function t_q^1 maps the set of nodes S_{1k}^r into itself, then for all the elements of the domain and range the indices 1 and k are the same. Therefore each element $S_{1,k,m} \varepsilon S_{1k}^r$ may be uniquely identified by m. Henceforth, for the sake of notational convenience, we may use $t_q^1(m)$ to denote $t_q^1(S_{1,k,m})$ whenever there is no ambiguity.

As a second restriction on the set of Σ-structures SIG_L we state the following:

ii) For every level $1 \leq L$ of $\Sigma_L^r \varepsilon SIG_L$ there exists a set of traversal functions t_q^1 defined for that level 1 for all values of r.

We are now in a position to describe p-expressions and their satisfaction. An *atomic p-expression* is an expression of the form: (i<>j) where i and j are variable symbols. Its satisfaction requires the presence of a "structure"; we

shall deal with this shortly after we fully define p-expressions.

Definition 2.5

The well formed formulas of p-expressions (pwff) are defined as follows. Let wff denote the well formed formulas of first order predicate calculus[MAN 74] then a pwff is obtained by using the following rules.

1) An atomic p-expression is a pwff.
2) If A is a quantifier-free pwff, then ~(A) is a pwff.
3) If A is a pwff, then $((A)_{\Lambda}(B))$ and $((A)_{U}(B))$ are pwff's where B is either a pwff or a universal wff such that one of the following cases is satisfied. Case a) If A has two free variables, then the only free variables in B, if any, are those occurring in A. Case b)If there exists only one free variable, i, in A, then B may only have either two free variables one of which must be i, or one free variable, or none. Case c) If A has no free variables, then B may have at most two free variables.
4) If A is a pwff then $\forall_{\iota}(A)$ is a pwff where $\iota \geq 1$.
5) Only those formulas obtained by a finite number of applications of (1) to (4), are pwffs.[1]

■

---

[1] Note that p-expressions are universal formulas.

Since we will be concerned with only pwff's, henceforth the term "p-expression" implies a pwff. We shall also omit the parentheses where there is no ambiguity.

In general, in order to determine the satisfaction of a first order predicate expression we need to define a "structure"[END 72].[2] A structure V is a function whose domain is the set of parameters:

1- $\forall_1$, $\forall_2$, $\forall_3$,....one for each positive integer.

2- Predicate symbols: =, ≤, ≥, <, >, <>.

3- Constant symbols: $S_{1k\emptyset}$, S_{1k1},...,S_{1km}, S_{1kn}, ..

4- Function symbols: +, −, x, /, t_q, f, g, h. The arity of these functions is generally either 1 or 2 unless otherwise specified.

such that:

1) V assigns to each quantifier symbol $\forall_\iota$ a non-empty set $|v_\iota| \subseteq |V|$; where $|V|$ is called the *universe* of V.

2) V assigns to each n-place predicate symbol P an n-ary relation $P^V \subseteq |V|^n$.

3) V assigns to each constant symbol c a member c^V of the universe $|V|$.

4) V assigns to each n-place function symbol f an n-ary operation f^V: $|V|^n \rightarrow |V|$.

---

[2] Some authors refer to this as an "interpretation."

Furthermore we need to define a function $s: Var \rightarrow |V|$ from the set Var of all variables into the universe $|V|$ of V. The extension of s denoted $\bar{s}$ is defined as follows:

1) For each variable i, $\bar{s}(i)=s(i)$.
2) For each constant symbol c, $\bar{s}(c)=c^{V}$.
3) If $t_1,\ldots,t_n$ are terms and f is an n-place function symbol, then
   $\bar{s}(ft_1,\ldots,t_n)=f^{V}(\bar{s}(t_1),\ldots,\bar{s}(t_n))$

The above definition of s maps all the variables into the whole universe $|V|$. This definition may be extended to those cases where there are different sets of variable symbols Var_1, $Var_2,\ldots,Var_N$ such that each set of variables VAR_ι may be mapped into some $|V_\kappa| \subseteq |V|$ by s.

Let us now present the definition of satisfaction of a wff, p, for the above structure where s maps different set of variables Var_ι to different subsets of the universe $|v_\iota| \subseteq |V|$. We say that V *satisfies* p *with* s, denoted by $\models_V p\ [s]$, as follows:

Let α and β denote a wff

1) For an n-place predicate parameter p
   $\models_V pt_1\ldots.t_n$ iff $<\bar{s}(t_1),\ldots.,\bar{s}(t_n)> \varepsilon p^{V}$
   where $t_1,\ldots,t_n$ are terms.
2) $\models_V \sim\alpha[s]$ iff $\not\models_V \alpha[s]$
3) $\models_V \alpha_\Lambda\beta[s]$ iff $\models_V \alpha[s]$ *and* $\models_V \beta[s]$
4) $\models_V \alpha_U\beta[s]$ iff $\models_V \alpha[s]$ *or* $\models_V \beta[s]$ *or both*

5) $\models_V \forall_l i\alpha[s]$ iff for every $d\varepsilon|v_l|$ $\models_V \alpha[s(i|d)]$ Here $s(i|d)$ is the function which is exactly like s except for one thing. At the variable i it assumes the value d.

To define satisfaction of a p-expression p_1, we need to define the type of structures that we shall be dealing with. Let S be the set of all possible nodes S_{lkm}, where l, k and m are non-negative integers. Let $S_{lk} \subseteq S$, denote the set of nodes, S_{lki}, having the same values of l and k. Thus $S_{lk}=\{S_{lki}: i\varepsilon \mathbf{N}^+\}$, where $\mathbf{N}^+$ is the set of non-negative integers. It is obvious that the set S_{lk} is isomorphic to $\mathbf{N}^+$. Because of this, in cases where the values of l and k are known, we may refer to the nodes as simply: 0, 1, 2,..., m, n,..., where we really mean $S_{lk\emptyset}$, S_{lk1}, S_{lk2},..., S_{lkm}, S_{lkn},.... A structure σ^r_{lk} is defined for each $\Sigma^r_L \varepsilon SIG$ for each level l and for each k. The universe (domain) of each structure σ^r_{lk} is S_{lk}. In every structure, there exists a distinguished family of traversal functions t^l_q, $1 \le q \le m_l$, such that each t^l_q maps $S_{lk} \to S_{lk}$. Such a structure may contain other functions and relations depending on the nature of the p-expression employed. In general any computable function and decidable relation may be defined in a structure σ^r_{lk}. It should be noted that for every structure σ^r_{lk}, the set S^r_{lk} is a subset of its universe S_{lk} representing the set of nodes "present" in the structure. All the variable symbols i,j, lki, lkj,....that are used in a pwff or any other wff

may only map into the set $S^r_{1k} \subseteq S_{1k}$ by the appropriate assignment function s.

Let us call the structure described above σ^r_{1k}. From now on, unless otherwise stated, σ^r_{1k} assigns to $\forall_1$ the set S^r_{1k} which is a subset of the universe $|\sigma^r_{1k}|=S_{1k}$.

In the following definition of satisfaction, we have only defined the satisfaction for the cases where the only quantifier allowed is $\forall_1$ (abbreviated as $\forall$). Henceforth we shall restrict the pwff's to those quantifiers whose domains are finite.

Let p_1 be a pwff then σ^r_{1k} *satisfies* p_1 *with* s, denoted by $\models_{\sigma^r_{1k}} p_1[s]$, as follows.

1) $\models_{\sigma^r_{1k}} (i<>j)[s]$ iff $s(i), s(j) \varepsilon S^r_{1k}$ and $s(j)=t^1_q(s(i))$ for some $1 \le q \le m_1$ where $s(i)$ and $s(j) \ne S_{1k\emptyset}$.[3]

Let α and β denote either a wff or a pwff

2) For an n-place predicate parameter p

   $\models_{\sigma^r_{1k}} pt_1 \ldots t_n$ iff $<\bar{s}(t_1), \ldots, \bar{s}(t_n)> \varepsilon p^V$

   where $t_1, \ldots, t_n$ are terms.

3) $\models_{\sigma^r_{1k}} \sim\alpha[s]$ iff $\not\models_{\sigma^r_{1k}} \alpha[s]$

4) $\models_{\sigma^r_{1k}} \alpha_\Lambda\beta[s]$ iff $\models_{\sigma^r_{1k}} \alpha[s]$ *and* $\models_{\sigma^r_{1k}} \beta[s]$

5) $\models_{\sigma^r_{1k}} \alpha_U\beta[s]$ iff $\models_{\sigma^r_{1k}} \alpha[s]$ *or* $\models_{\sigma^r_{1k}} \beta[s]$ *or both*

6) $\models_{\sigma^r_{1k}} \forall i\alpha[s]$ iff for every $m \varepsilon S^r_{1k}$, $\models_{\sigma^r_{1k}} \alpha[s(i|m)]$ Here $s(i|m)$ is the function which is exactly like s except

---

[3] Note that t^1_q denotes the function t^V, where t is a traversal function symbol and V, in here, is of course σ^r_{1k}.

for one thing. At the variable i it assumes the value m.

__Example__

Consider a structure σ_{1k} with traversal functions $t_1(i)=2\times i$ and $t_2(i)=2\times i+1$.[4] Assume a p-expression: $i<>j$. Let $s(i)=<1,1,2>$ and $s(j)=<1,1,5>$ then since $5=2\times 2+1$, the nodes indexed <1,1,2> and <1,1,5> are adjacent (or u-related) because the above p-expression is satisfied with s. However if $s(i)=<1,1,2>$ and $s(j)=<1,1,9>$ then the p-expression is not satisfied with s and therefore the nodes indexed <1,1,2> and <1,1,9> are not u-related (or not adjacent).

Let us now look at a slightly more complicated example of a p-expression which may be formed using the above definition of pwff:

__Example__

Let $S^r_{11}=\{<1,1,1>,<1,1,2>,<1,1,3>,<1,1,4>,<1,1,5>\}$ and $t^1_q(S_{1ki})=S_{1ki+1}$. Let $p(i,j)=(i<>j)_\Lambda(i\leq 10)$; and $s(i)=<1,1,3>$ and $s(j)=<1,1,4>$. The relation $\leq$ is defined as follows: $S_{1,k,m}\leq^{\sigma^r_{11}}S_{1k\mu}$, if m is less than or equal to μ. With s as defined above we have:

---

[4] In here we have extended the usual definitions of $\times$ and + such that $\kappa\times<1,k,i>=<1,k,\kappa\times i>$ and $\kappa+<1,k,i>=<1,k,i+\kappa>$ where κ is an integer. As we mentioned before since 1 and k are fixed here, we are only concerned with the node index i. Hence every node may be treated just like an integer.

$(\langle 1,1,3\rangle <> \langle 1,1,4\rangle) \wedge (\langle 1,1,3\rangle \leq \langle 1,1,10\rangle)$

The expression $(\langle 1,1,3\rangle \leq \langle 1,1,10\rangle)$ is obviously true. To know if $\langle 1,1,3\rangle<>\langle 1,1,4\rangle$ expression is satisfied for the given s we refer to the definition of satisfaction of an atomic p-expression given earlier (i.e. if $s(i),s(j) \varepsilon S^r_{lk}$ and $s(j)=t^l_q(s(i))$ then it is satisfied else not satisfied). Substitute $\langle 1,1,3\rangle$ for i and $\langle 1,1,4\rangle$ for j; thus we have $t^l_q(\langle 1,1,3\rangle)=\langle 1,1,3+1\rangle=\langle 1,1,4\rangle$ which is equal to $s(j)=\langle 1,1,4\rangle$. As a result the p-expression p(i,j) is satisfied with the above s. Clearly there are other values of i and j that would satisfy p(i,j), whereas some pairs such as $\langle 1,1,2\rangle$ and $\langle 1,1,5\rangle$ would result in the p-expression not satisfied since $5 \neq 2+1$. Hence the satisfaction depends on what the values of s(i) and s(j) are.

Let us take one step further and deal with a whole class of Σ-structures, **SIG**, such that the same family of p-expressions may be used to describe u-relations, at different levels, for every element of **SIG**.

Definition 2.6

Let þ be a family of p-expressions $þ=\langle p_1,\ldots,p_L\rangle$. Let SIG_L be a set of Σ-structures such that it obeys the restrictions (i) and (ii) with a *finite* number, m_l, of traversal functions t^l_q, at every level l. Let σ^r_{lk} be a structure as defined before equipped with the traversal functions t^l_q and any other computable functions or decidable

relations. Then SIG_L is *implementable with* $\mathfrak{p}$ if for every r and every level l and every k, and any assignment function s, the following is true.

1) If $\pi(i,j)$ is the p-expression at level l such that i and j are both free variables, then
$\models_{\sigma^r_{lk}} \pi(i,j)\ [s_l]$ iff[5] $(s_l(i),s_l(j))\varepsilon u^r_{lk}$.

2) a- If $\forall i\pi(i,j)$ is the p-expression at level l and j is the only free variable in π, then
for every $d\varepsilon S^r_{lk}$, $\models_{\sigma^r_{lk}} \pi(i,j)\ [s_l(i|d)]$ iff
for every $d\varepsilon S^r_{lk}$, $(d,s_l(j))\varepsilon u^r_{lk}$

2) b- If $\forall j\pi(i,j)$ is the p-expression at level l and i is the only free variable in π, then
for every $d\varepsilon S^r_{lk}$, $\models_{\sigma^r_{lk}} \pi(i,j)\ [s_l(j|d)]$ iff
for every $d\varepsilon S^r_{lk}$, $(s_l(i),d)\varepsilon u^r_{lk}$

3) If $\forall i\forall j\pi(i,j)$ is the p-expression at level l, then
for every d_1, $d_2\varepsilon S^r_{lk}$, $\models_{\sigma^r_{lk}} \pi(i,j)\ [s_l(i|d_1)(j|d_2)]$
iff for every d_1, $d_2\varepsilon S^r_{lk}$, $(d_1,d_2)\varepsilon u^r_{lk}$

p_l is referred to as the p-expression *associated with* level l of SIG_L

Let SIG_L be implementable and $\Sigma^r_L=(S^r,u^r)\varepsilon SIG_L$; let t^l_q, $1\le q\le m_l$, be a traversal function for level l of every Σ^r_L. Then Σ^r_L may be described by $(S^r,\mathfrak{p})$ where $\mathfrak{p}=\langle p_1,\ldots,p_L\rangle$ since from the above definition, for every $l\ge1$, u^r_l may be defined

---

[5] s_l is the assignment function at level l; such that s may be considered to be a family of s_l for every level l, $1\le l\le L$.

by the associated p-expression. Therefore it would suffice to have just the pair (S^r, p) to describe (S^r, u^r).

Definition 2.7

Let S^r be the set of nodes in a Σ-structure $\Sigma_L^r=(S^r,u^r)$, A node $S_{lkj}\varepsilon S^r$ is a *successor* of a node $S_{lki}\varepsilon S^r$ denoted $j=suc(i)$.

1) if $(i,j)\varepsilon u_1^r$ and $j\geq i$, or

2) if $j'=suc(i)$ and $(j,j')\varepsilon u_1^r$ and $j\geq j'$.

We also say that i is a *predecessor* of j, i=pred(j), if j=suc(i). The suc(Ø)=pred(Ø)=Ø. Finally, j=1_suc(i) if $(i,j)\varepsilon u_1^r$.

Definition 2.8

A *successor set* of a node $S_{lki}\varepsilon S$ is:

$$sucset(i)=\{ j : j=suc(i) \}.$$

Definition 2.9

Two Σ-structures $\Sigma=(S,u)$ and $\Sigma'=(S',u')$ are *isomorphic*, denoted $\Sigma\simeq\Sigma'$, if there exists a bijection η: $S\rightarrow S'$ such that $(i,j)\varepsilon u$ iff $(\eta(i),\eta(j))\varepsilon u'$.

For a family of Σ-structures SIG_L, it is imperative that it can be decided whether an arbitrary Σ-structure is an element of SIG_L. One way to decide such a membership is

to show that the problem of satisfaction of p-expressions is decidable. This is demonstrated below.

Lemma 2.1

Let $\Sigma=(S,u)$ be a Σ-structure; let $S_{1k} \subseteq S$ and σ_{1k} be a structure as defined earlier; let p be a pwff such that the only variables in p are those denoting the elements of S_{1k}. Then it is decidable whether $\models_{\sigma_{1k}} p[s]$, for any given s, or not.

Proof

Assume p is written in prenex normal form(pnf), such that it may be represented in the form of i<√>j#Ξ with or without quantifiers in front of it, where i<√>j denotes either i<>j or ~i<>j and # is either an AND or an OR function. Note that this is possible since pwff's are special cases of first order wff's. And since every wff may be written in pnf, therefore every pwff may be written in pnf. Recall that the "formula-building operators" of pwff's are the same as those of wff's(see definition 2.5(2)-(3)). Also Ξ denotes a wff such that the p-expression is in fact a pwff. Consider the following cases (a)-(d) where Ξ is assumed to be a wff. We shall show the satisfaction by induction on the number of prime formulas. Cases(a)-(d) show the basis of this induction. The extension to the case of n prime formulas is trivial as described in each case that follows.

a) p is either $i<>j$ or $\sim i<>j$. Since the number of traversal functions is finite, therefore $\models_{\sigma_{1k}} p[s]$ or its negation may be decided in a finite number of steps.

b) p is $p_1\#\Xi$, where $p_1=i<\surd>j$. Ξ is a quantifier-free wff. Hence it may be written in the pnf as follows:

$p_1 \#_1 p_2 \#_2 \cdots\cdots \#_n p_n$.

where each p_q, $1\leq q\leq n$, is a predicate with either zero or one or two free variables. Furthermore each p_q is quantifier-free and assigned, by the structure σ^r_{1k}, to some decidable relation. There are three cases to consider:

1- p_q	(no free varibales occur in p_q)
2- $p_q(i)$	(one free variable)
3- $p_q(i,j)$	(two free variables)

Case (1) is trivially either *true* or *false*. Cases (2) and (3): $p_q(i)$ and $p_q(i,j)$ are assigned to some unary and binary relations respectively by the structure at hand. In both case, since the relations are decidable,[6] one can immediately find if i or (i,j) is an element of the corresponding relation or not. This is of course possible using the characteristic function of the relation. Hence for each p_q, $1\leq q\leq n$, we can decide if it is satisfied or not.

c) p is of the form $\forall i(i<\surd>j)\#\forall i\Xi$. That is either i is free and j is quantified over S_{1k}, or vice versa. Since S_{1k} is finite then $\forall i(i<\surd>j)$ is indeed decidable.

---

[6] A relation is (primitive) recursive if it is equipped with a (primitive) recursive characteristic function.

To decide the satisfaction of $\forall i \Xi$, we need to consider, for every $d \varepsilon S_{1k}$, the satisfaction of $|=_{\sigma_{1k}} \Xi \ s[i|d]$. This reduces the problem to that of (b) above. Clearly the case when i is free and j is bound is similar.

finally,

d) Both i and j are bound, i.e. p is of the form

$$\forall i \forall j (i<\rho>j) \# \forall i \forall j \Xi$$

Once again the problem reduces to that of: for every $d_1 \varepsilon S_{1k}$, $|=_{\sigma_{1k}} \forall i \Xi \ s[j|d_1]$. Thus the problem is the same as that of part (c) above.

e) All the other cases may be reduced to one or more of the above. This is shown below.

i) Consider the case of a pwff written in the form of $p=i<\rho>j\#\Xi$ where p is quantifier-free and Ξ is itself also a pwff. Then p may be written in the form of:

$$p=i<\rho>j \ \# \ i<\rho^1>j \ \#_1 \ \Xi^1$$

where $\Xi=i<\rho^1>j \ \#_1 \ \Xi^1$, and Ξ^1 is a quantifier-free pwff. Continuing with the above procedure, p may be decomposed as follows:

$$p=i<\rho>j \ \# \ i<\rho^1>j \ \#_1 \ldots\ldots \#_{n-1} \ i<\rho^n>j \ \#_n \ \Xi^n$$

where Ξ^n is either a wff or simply $i<\rho^m>j$. Now if the satisfaction of $p_{n-1}=i<\rho^1>j \ \#_1 \ldots\ldots \#_{n-1} \ i<\rho^n>j$ is decidable, then obviously the satisfaction of $p_{n-1}\#_n\Xi^n$ is also decidable by part (a) or (b) or both described above.

ii) Consider the case (i) above, but the occurrence of the quntifiers is also permitted. Using the definition 2.5(3)-(4), restricting ourselves to $\forall$ only, we can write a

general p-expression of the form shown below where the only free variables are i and j.

$$\forall k_1 \forall k_2 (k_1 <> k_2 \#_1 \Xi^1) \#_{11} \forall k_3 ((k_3 <\surd> j) \#_2 \Xi^2) \#_{22} \ldots . ((i <\surd> \#_n \Xi^n)$$

Let us only consider the case of $\forall k_1 \forall k_2 (k_1 <\surd> k_2 \ \#_1 \ \Xi^1)$. This would show the basis of the induction. First we assume that Ξ^1 is a quantifier-free pwff. WE may write the above expression in the form of $\forall k_1 \forall k_2 (k_1 <\surd> k_2) \ \#_1 \ \forall k_1 \forall k_2 \Xi^1$. But the satisfaction of $\forall k_1 \forall k_2 (k_1 <\surd> k_2)$ is decidable by the case (d) above. Therefore we are only interested to know if $\forall k_1 \forall k_2 \Xi^1$ is satisfied with some s, or not. This problem reduces to that of deciding the satisfaction of $\Xi^1 \ s[k_1|d_1][k_2|d_2]$ for every d_1, $d_2 \varepsilon S_{1k}$. But this is also decidable from (i) above since Ξ^1 is quantifier-free. Finally consider the case where Ξ^1, in the above expression is not quantifier-free. If k_1 and k_2 do not occur in Ξ^1, then the problem would become a special case of what follows. If one or both occur free in Ξ^1, then the problem is to decide: if for all d_1, $d_2 \varepsilon S^r_{1k}$, $\Xi^1 \ s[k_1|d_1][k_2|d_2]$ is satisfied or not.

If there are more occurrences of the $\forall$ quantifier in Ξ^1, viz. $\forall k_l$, we proceed in the same way as above until all the $\forall$ quantifiers are removed. The latter form would then lend itself to that of case (i) above.

Lemma 2.2

Let SIG_L be implementable wrt $\mathfrak{p}=\langle p_1,...,p_L\rangle$, then SIG_L is recursive.

Proof

First we consider the case for L=1 and $\mathfrak{p}=\langle p_1\rangle$, where $p_1=\pi(i,j)$ and i and j are free in π. Let SIG be the set of 1-level Σ-structures $\Sigma=(S,u)$. For SIG to be recursive we need to have a recursive characteristic function ϕ such that $\phi(\Sigma)=1$ if $\Sigma\varepsilon SIG$, and =0 otherwise.

Since u is finite we can examine each pair $(\mu,\nu)\varepsilon u$ to see if $\pi(\mu,\nu)$ is satisfied and vice versa, where μ, $\nu\varepsilon S$ and S is finite. But by lemma 2.1 the problem of deciding if π is satisfied, or not, is computable in a finite number of steps. Therefore ϕ can be realized as follows:

$\phi(\Sigma)=1$ if for μ, $\nu\varepsilon S$, $(\mu,\nu)\varepsilon u$ iff $\pi(\mu,\nu)$ is satisfied[7]

$\phi(\Sigma)=\emptyset$ otherwise.

For L>1 we can have a finite number of characteristic functions ϕ_l, one for each level l. Clearly the above argument holds for every S_{lk}, where for each ϕ_l, if $\phi_l((s_{lk},u_{lk}))$ evaluates to 1 for all S_{lk}, then $\phi(\Sigma)=1$, and =0 otherwise. Note that at every level the set S_l is the union of a finite number of finite sets S_{lk}.

---

[7] This may readily be extended to the cases where the p-expression has only one or no free variables.

A "data structure", as will be seen later, is an implementable set of Σ-structures. As an example consider a list structure, Σ, where the p-expression associated with it is i<>j, where the traversal function is $t(i)=i+1$. Let $\Sigma=(\{1,2,3\}, <(1,2),(2,3)>)$. Using the above definition of the characteristic function we can see that $\phi(\Sigma)=1$. Hence Σ is an element of the set of all list structures.

Corollary 2.1

Let SIG_L be implementable with þ. SIG_L is recursively enumerable(r.e.).

Definition 2.10

The *largest* set of Σ-structures, SIG_L, implementable wrt þ, and satisfying the following conditions is called an L-level *data structure* Z_L. For every $\Sigma_L^r=(S^r,u^r)\varepsilon Z_L$, and every level l and for every k, if i, $j\varepsilon S_{lk}^r\neq\{S_{lk\emptyset}\}$, then*

$$(i> MIN(S_{lk}^r)\rightarrow(\exists j(j<i_\Lambda(j,i)\varepsilon u_{lk}^r)))\ldots.(a)$$

Condition (a), in the above definition, ensures that no node, except the smallest node S_{lki}(i.e. having the smallest "i-index"), is present without the presence of at least one of its 1-predecessors.

---

* Note that i and j in the above equation denote <lki> and <lkj> respectively. MIN(X) returns the element in X with the lowest index value other than $S_{lk\emptyset}$. Also $S_{l,k,i}>S_{l,k,j}$ is true iff i>j.

A "data structure configuration" is defined next.

Definition 2.11

For a given data structure Z_L, an element $z \varepsilon Z_L$ is a *data structure configuration*.

In every L-level data structure, the configuration $(\{S_{L1\emptyset}\}, \text{þ})$ is referred to as the *empty* or the *starting* configuration, denoted by ϕ. The empty configuration implies that although a data structure is already defined but no nodes have been "inserted" into the structure.

Lemma 2.3

Let Z_L be a data structure, then $\phi=(\{S_{L1\emptyset}\}, \text{þ}) \varepsilon Z_L$.

Proof

By the definition of data structures, Z_L is the largest set of Σ-structures which is implementable wrt þ. Therefore $\phi \varepsilon Z_L$ otherwise $Z_L \cup \{\phi\}$ is the largest set.

Lemma 2.4

Let Z_L be a data structure. Z_L is r.e.

Proof

The proof is essentially the same as that of lemma 2.2 and corollary 2.1 except that the condition (a) of definition 2.10 is also imposed on equation (1) of the proof. That is for $\phi(\Sigma)=1$, we must have the condition: for

every $j \varepsilon S^r_{1k}$, except if j is the smallest one, there exists an $i \varepsilon S^r_{1k}$ and $(i,j) \varepsilon u^r_{1k}$ iff $\pi(i,j)$ is satisfied. Note that the condition (a) is decidable since S^r_{1k} is finite.

Example

Consider a two level data structure , namely **tree-of-lists.** Structures (a), (b), and (c) depicted in figure 2.3 are valid configurations, whereas (d) is not, since a lower level structure cannot exist without its immediate higher level(supporting) structure. This of course follows from the definition of Σ-structure.

The family of p-expressions Ꝑ is $<p_1, p_2>$, where

p_2: $(i<>2i) \cup (i<>2i+1)$

p_1: $i<>i+1$

Our notion of data structures do not completely specify the semantics of a data type. It only specifies the relationship between the "data items." In order to completely characterize the "behaviour" of a data type, we need operations to be performed on data structures. A pair, data structure and its operation set, yields a "data type." A preliminary and informal definition of a data type is given below. A more rigorous definition will be presented in the next chapter after we define the "Data Structure Operations." A *data type*, t, is a pair (Z_L, O) where Z_L is a

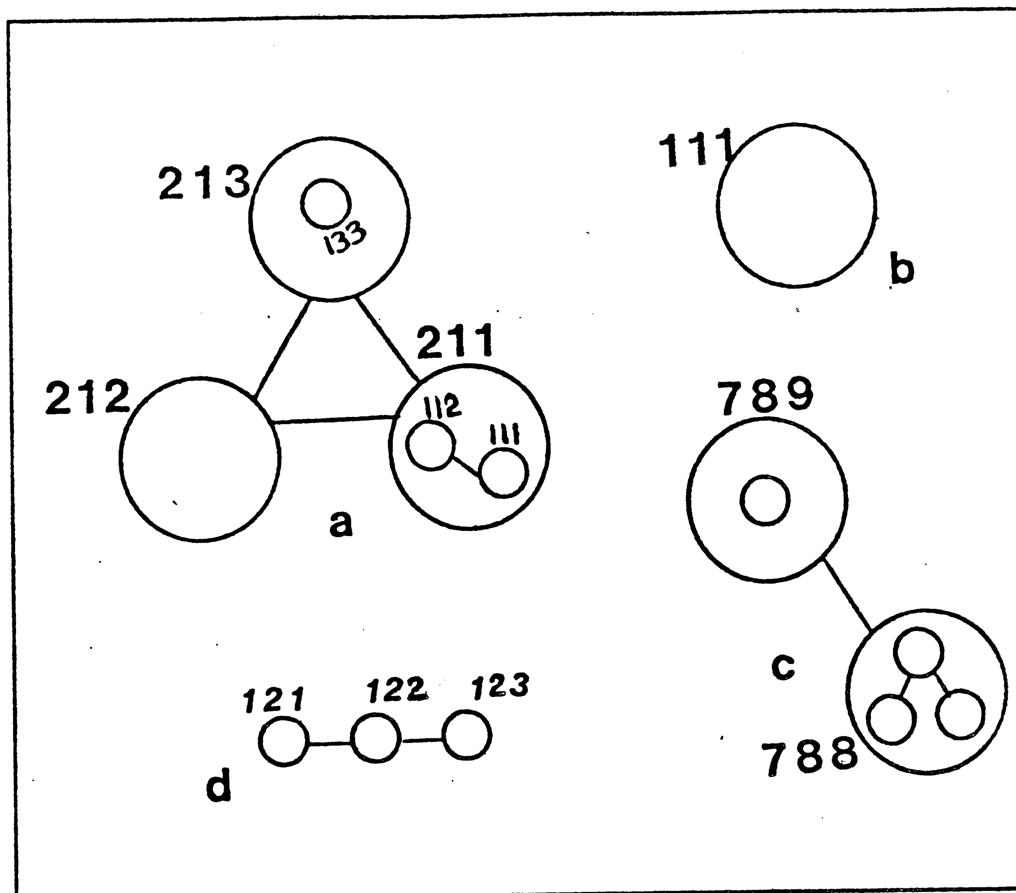


Figure 2.3
Data Structure Configurations.

L-level data structure, and O is a set of operations which enable us to make transitions between elements of Z_L.

The set O will be referred to as the set of *Data Structure Operations(DSO)*. It should be noted that for a given data type, the number of levels L of its data structure remains unchanged under any DSO operations.

Chapter 3 identifies the nature of the DSO operations and a minimal set of operations which would universally

specify data types. We shall see that, in the presence of such universal operators, some "fine tuning knobs" made available to the user would suffice in order to specify the particular behaviour which is desired.

CHAPTER III

DATA STRUCTURE OPERATIONS

The behaviour of abstract data types is generally determined by the operations defined on them. We shall refer to such operations as data structure operations. A major problem has always been to define a set of operations in order to completely specify some desired behaviour. The use of logical axioms and equations to specify data structure operations(DSO) have been most popular in the last decade. In almost all cases however, the techniques deploying the axiomatic approach, have left the user with some degree of uncertainty regarding the completeness and consistency of the specification. Such problems are common even in the case of simple and reasonably well understood data types.

Data structure operations may, in general, be divided into two categories: the characteristic set and the auxiliary set of operations. The former characterizes the behaviour, hence it includes the constructor operations and operations concerning the access of data entities. The auxiliary set contains operations that are not crucial to the behaviour exhibited by the data type. For example in a

stack, push, pop, and top are all elements of the characteristic set. The auxiliary set of operations for **stack** may contain operations such as is_empty or is_full. It should be noted that the latter operations may all be derived from the elements of the characteristic set, a process commonly referred to as "enrichment."

In this chapter a universal DSO set will be presented. These operations are employed to define the characteristic set of operations of a specific data type with the aid of three types of first order predicate expressions.

Notation: Throughout this chapter for notational brevity as well as clarity we have referred to nodes mostly by their third index value m. In such cases the values of l and k are either assumed or the argument presented may readily be extended to nodes having different l and/or k values. Also implicit in some of our discussions, will be the levels of the underlying structure. In many instances the arguments are presented either for level 1 only of an L-level Σ-structure or simply a 1-level structure is assumed. Once again unless otherwise stated, we may readily extend the arguments to the general case. Unless otherwise specified all the Σ-structures terminate with SSN. In order to identify the traversal functions at a given level, we employ the abbreviation described below. This would enable us to express both the p-expression as well as the traversal functions needed to decide the satisfaction of the p-expression. If we assume that the underlying structure, σ,

has the traversal functions t_1, $t_2,\ldots,t_n$ associated with it, then

- $i<>t_1(i) \cup \cdots \cup i<>t_n(i)$ replaces every occurrence of $i<>j$ in the p-expression.
- $i<\not>t_1(i) \cup \cdots \cup i<\not>t_n(i)$ replaces every occurrence of $\sim i<>j$ in the p-expression.

Using the above notation, we can explicitly state the actual traversal functions associated with every level as well as specifying the p-expression at that level. As a final note, let $\alpha(i)$ be a first order predicate formula; we say $\alpha(m)$=*true* in the presence of some structure σ iff σ satisfies $\alpha(i)$ with any assignment function $s[i|m]$; $\alpha(m)$=*false* iff $\alpha(i)$ is not satisfied with $s[i|m]$.

3.1- <u>Characteristic Set of Operations</u>

Every data type is required to have a set of operations defined over its data structure in a manner which characterizes the intended behaviour of the data type. For example we are not allowed to delete a node from the middle of a **stack** data type. As a result insertion and deletion operations must ensure that no such illegal operations are carried out. We shall see that due to the way we define DSO's the "correct" configurations will always result. Furthermore, the p-expressions may be used in order to assert the correctness of the DSO operations by Floyd/Hoar approach to program correctness. For example p-expressions may be employed as pre- and post-conditions before and after

every data structure operation respectively. Consequently we would ensure that the structure remains intact. In general, from an abstract point of view, we are interested in developing some common characteristics exhibited by the DSO's of all data types. We shall present certain requirements to be possessed by the elements of a characteristic DSO Set.

It will be shown that we can form all the configurations of a data type using the "characteristic DSO set." This philosophy is similar to the notion of the canonical algebras and the constructor signature approach of Goguen et. al., although we pursue a more general approach.

Before defining the elements of the characteristic DSO (CDSO) set, the notion of "b-expression" is introduced. These predicate expressions will be used to define "behaviour." Hence illegal insertion and deletion of nodes in a structure are avoided.

A b-expression is a first order predicate calculus expression whose function is to determine the "boundary conditions" of an insertion or a deletion function defined for a data structure. For example insertion into a full-stack violates the boundary conditions of the push function; deletion from an empty-stack is also inconsistent with the boundary conditions of the pop function. Insertion into the middle of a FIFO queue is another example of an erroneous operation.

The index of the node to be inserted(or deleted) has to meet certain conditions. Such conditions are specified by the b-expressions associated with the data structure. A b-expression, b(i), defines the range of values that the node index can take for insertion (deletion) purposes. Thus if m is within the desired range, then b(m)=*true*; b(m)=*false* otherwise. Clearly the satisfaction of a b-expression, defined below, depends upon the underlying structure. Consider a **stack** of depth 4. A b-expression associated with this stack may, or may not, permit further insertion into the stack depending on what the maximum length of the **stack** is supposed to be. Let us assume that the maximum permitted length is 5. After one more insertion into the stack of length 4, the b-expression would have to prevent any further insertions. Thus the b-expression associated with a class of structures must be able to dynamically convey, to the system, sufficient information for insertion and deletion purposes. Intuitively, a b-expression, b, defines the places where insertion (deletion) of a node is, or is not, allowed.

There is always a possibly empty set of nodes that may be inserted into a data structure configuration without disturbing the intended behaviour of the data type. Such a set varies with different configurations of a data structure. Similarly, there always exists a finite set of nodes that may be subjected to deletion. Hence there exists a variable set from which deletions can be made with respect

to the configurations of a data structure. More precisely, consider a configuration of a data structure z=(S,u) where S is the union of all S_{1k}, then we may define a "construction set" for each (S_{1k},u_{1k}) of z as follows. The *construction set* of S^r_{1k}, CS^r_{1k}, is a subset of the *total construction set* (TCS) for S^r_{1k} defined below. Let m denote an element of $|\sigma^r_{1k}|$ and n denote an element of S^r_{1k}, then

$TCS^r_{1k}=\{m: \models_{\sigma^r_{1k}} p_1$ with either s[i|m][j|n] or s[j|m][i|n]$\}$'

where s is any assignment function and $S^r_{1k}\neq\{S_{1k\emptyset}\}$. If $S^r_{1k}=\{S_{1k\emptyset}\}$ then,

$$TCS^r_{1k}=\{S_{1km} : m=MAX(\bigsqcup_{q<k} S_{1q})+1\}$$

where $MAX(\bigsqcup S_{1k})=j$ where $S_{1kj}\varepsilon\bigsqcup S_{1k}$ and for all $S_{1ki}\varepsilon\bigsqcup S_{1k}$, $j\geq i$; also $MAX(\{<1k_1\emptyset>,<1k_2\emptyset>,..,<1k_n\emptyset>\})=\iota$, $k_r\geq\emptyset$; and $MAX(\{\})=\iota$ where ι is some integer greater than zero.

Corollary 3.1

For a given S^r_{1k}, the set TCS^r_{1k} is recursive.

Proof

We can define a characteristic function ϕ such that $\phi(m)=1$ if $m\varepsilon TCS^r_{1k}$ and $\phi(m)=0$ otherwise. To show that $m\varepsilon TCS^r_{1k}$, we need to demonstrate the satisfaction of:

$\models_{\sigma^r_{1k}} p_1$ s[i|m][j|n] *or* $\models_{\sigma^r_{1k}} p_1$ s[j|m][i|n]

' i and j are the two free variables, if any, that may occur in a p-expression p_1.

By lemma 2.1, this problem is decidable. Hence the characteristic function ϕ is computable. Therefore, TCS^r_{1k} is recursive.

Trivially, a total destruction set of a S^r_{1k} is the set S^r_{1k} less the empty node $S_{1k\emptyset}$. But more interesting is the set of nodes permitted for deletion from a given configuration, the latter will be referred to as the *destruction set*(DS). The "b-expression for deletion" would define the exact subset of S^r_{1k} which is equal to the DS^r_{1k}. Let us now define a "b-expression" for a data structure Z. In chapter 2 we pointed out that a data structure Z may be equipped with a set O of "insertion" and "deletion" as well as some other functions of interest. The insertion and deletion functions enable transitions from one configuration of Z to any other configuration, in Z, by addition or subtraction of nodes. The b-expressions help us to determine the domain of these functions.

Definition 3.1

Let wff denote a well formed formula of first order predicate calculus. Then a *b-expression well formed formula*, denoted as *bwff*, is defined as a universal wff in which there exists exactly one occurrence of a free variable.

We shall deal with two types of b-expressions for every level l of a data structure: b-expression *for deletion*, b_{δ}^{l}, and b-expression *for insertion*, b_{ι}^{l}.

The satisfaction of the b-expressions is essentially the same as that of pwff's. Consider a structure σ_{lk}^{r} as defined in chapter 2. Let the range of values of the free variable, i, be $TCS_{lk}^{r} \subseteq |\sigma_{lk}^{r}|$, and the range of values of all the other variables be S_{lk}^{r}. If TCS_{lk}^{r} is empty then the b-expression is not satisfied with any s. Recall that $\forall$ refers to "for all elements in S_{lk}^{r}."

Definition of the satisfaction of b_{δ}^{l} is exactly the same as that of b_{ι}^{l} above, except that s maps all the variables, including the free variable i, to S_{lk}^{r}.

Now using the two b-expressions mentioned above, we may define the two sets: the construction set and the destruction set for S_{lk}^{r}:[10]

$$CS_{lk}^{r}=\{i : b_{\iota}^{l}(i)\}$$

$$DS_{lk}^{r}=\{i : b_{\delta}^{l}(i)\}$$

For every data structure we associate a b-expression b_{ι}^{l} (b_{δ}^{l}) for insertion (deletion) at level l, for all levels $l \leq L$. Then for every level, we may associate CS's and DS's for every S_{lk}^{r} at that level using the above definition.

---

[10] $A=\{i : \alpha(i)\}$ denotes that for every value of i, say m, that $\alpha(i)$ is satisfied with, then $m \varepsilon A$.

Lemma 3.1

Given a structure σ^r_{1k}, it is decidable whether b^1_ι (b^1_δ) is satisfied, or not, for a given s.

Proof

The proof is the same as that of lemma 2.1. Note that here we have added a domain, TCS^r_{1k}, to our universe of the structure. However, since there are no quantifiers over TCS^r_{1k}, therefore the satisfaction problem is decidable.

Using the above result, it becomes a trivial task to determine whether for a given structure, (S_{1k}, u_{1k}), in a configuration of a data structure, a certain node index is, or is not, in its construction set. Let Z_L be a data structure and let $b^1_\iota(i)$ ($b^1_\delta(i)$) denote the associated b-expression for insertion (deletion) at level 1 of Z_L. Consider now a set S^r_{1k} at level 1. Define the characteristic function for CS^r_{1k} as follows:

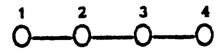
$$\phi(m)=1 \quad \text{iff} \quad b^1_\iota(m)=\mathit{true}$$
$$=0 \qquad \text{otherwise}$$

Since the satisfaction of the b-expression is decidable for a given s, ϕ is computable. In other words for every S^r_{1k}, CS^r_{1k} (DS^r_{1k}) is a recursive set. It also follows that CS^r_{1K} is r.e.

Example

Consider a configuration of the list data structure

S_{11}, where the p-expression is i<>i+1:[11]



The question is whether a node indexed <1,1,675> (say) is an element of CS_{11}? This is not the case since there exists no $i \varepsilon S_{11}$ such that i+1=675 or 675+1=i, see definition of TCS. Hence <1,1,675> is not an element of TCS_{11}. Therefore $\phi(<1,1,675>)=0$. Now consider a node <1,1,5>, abbreviated as 5; by the same argument we would find that $\phi(5)=1$. Hence $5 \varepsilon CS_{11}$. Finally consider a **list** data structure where the length of its elements must be less than or equal to four nodes. Such a restriction, on the length of the configurations, is imposed using the b-expression for insertion, viz. $b_\iota^1 \equiv i \leq 4$. Then although 5 is an element of TCS_{11}, $5 \notin CS_{11}$, since the b-expression for insertion, $i \leq 4$, would yield $b_\iota^1(5)=$*false*.

Before we define functions for insertion and deletion of nodes it is necessary to introduce the notion of accessibility of nodes. This is essential since insertion and deletion functions must be restricted only to the structures that are contained in "accessible" nodes.

---

[11] Note that for all the nodes <lki>, the value of l and k is fixed, i.e. (l,k)=(1,1). Hence a node may be referred to by its third index i. Henceforth we may write relations such as $<lki> \geq \iota$ where S_{lki} is a node and ι is an integer. This relation merely implies $i \geq \iota$. Same approach may be adopted in case of functions; for instance $S_{lki}+\iota$ denotes $S_{lki+\iota}$.

The issue of data access and security has gained considerable importance in the study of data base systems as well as the operating systems. In order to incorporate this concept into our model, another function of crucial importance is introduced, namely the probe function. The purpose of a probe function, as suggested by its name, is to enable the user to have access to various nodes in the structure. However an important issue is whether every node is permitted for access or not. In the **stack**, for example, the only node which is permitted for access is the top node. This may also be the case in a **queue** data type where the "front" node is the only node accessible. In contrast in an **array** data type one may probe any one of the nodes in the structure.

An abstraction of the notion of the accessibility of nodes may be carried out by means of a certain kind of predicate expressions described below. With every data structure, Z_L, we may associate a family of first order predicate formulas, namely the a-expressions. An a-expression defines the set of nodes which may be accessed in every configuration of that data structure.

The well formed formulas of the a-expressions, denoted *awff*, are the same as bwff's. The definition of their satisfaction in the presence of a structure σ_{1k}^{r} (defined earlier in chapter 2) is exactly the same as that of b_{δ}^{1} with

the following exception. Let $\alpha_l(i)$ denote the a-expression at level l, then σ_{lk} satisfies $\alpha_l(i)$ with s if

- $|=_{\sigma_{lk}} \alpha_l(i)s$ AND if $l<L$ then $\sigma_{l+1,q}$ satisfies $\alpha_{l+1}(j)$ with $s[j|S_{l+1,q,k}]$ for some q.

For every level l of a data structure Z_L we associate an a-expression, $\alpha_l(i)$. The above definition of satisfaction of the a-expressions implies that, to access a node at level l, one must have access to all the other nodes at levels l+1 and above that contain the desired node.

We are now in a position to define the three types of functions that comprise the "characteristic data structure operation set." The i-function is defined first followed by the d-function and the p-function respectively.

Definition 3.2

Let $z=(S,u)$, $z'=(S',u')\varepsilon Z_L$, where Z_L is a data structure and let b_ι^l be the b-expression for insertion, and α_l be the a-expression, associated with level l, $1\le l\le L$. A *primitive insertion function* (i-function) is a partial mapping:

$$\iota \quad : Z_L \times N^3 \rightarrow Z_L$$

where $\iota(z, l, k, m)$ is *defined* and equal to z' if the following conditions hold at level l; let $k,m\ge 1$:[12]

---

[12] This equation denotes: insert a node indexed m at level l of the configuration z in node k of the (l+1)th level. The resulting configuration is z'; and the inserted node is labeled S'_{lkm}. Note that, the primed sets and symbols refer to the new configuration. Double primed symbols are merely "temporary" variables.

1) $b_l^1(i)$ is satisfied with σ_{lk} with $s[i|S_{lkm}]$,

2) If $l<L$ then $\sigma_{l+1,q}$ satisfies $\alpha_{l+1}(S_{l+1,q,k})$ for some q,

3) $S_{lkm} \notin S_{lk}$,

4) *If*[13] $S_{lkm} > MAX(S_{lk})$

then- Let $S"=S_U\{S_{lkm}\}$. Renumber the nodes at level l of S" starting with the structure contained in all the 1-successors of k and all the structures contained in nodes $S_{l+1,\kappa,q}$ for any κ such that q>k. The renumbering is done using the convention of definition 2.2, such that for every $S_{\lambda q}$, whose elements are renumbered to get $S"_{\lambda q}$, $(S_{\lambda q},u_{\lambda q}) \simeq (S"_{\lambda q},u"_{\lambda q})$. Note that the renumbering would not affect the l-index(level indicator) of the nodes. Also $S_\lambda = S"_\lambda$ for $1<\lambda \leq L$ and $S_{\lambda\kappa}=S"_{\lambda\kappa}$ for $\lambda=1$ and $\kappa<k$. S' is then assigned to be the same as S".

else- $S'=S_U\{S_{lkm}\}$.

5) $C(S'_{lkm})=\Sigma_{undefined}$ if l=1 & Σ terminates with SSN,
$\quad = S'_{l-1,m,\emptyset}$ otherwise.

<u>Example</u>

An i-function for the **stack**

$S_1 = \{S_{111}, S_{112}, S_{113}, S_{114}\}$

$p_1 = i<>(i+1)$

$i(z,1,1,5)=z'$

$S'_1 = \{S'_{111}, S'_{112}, S'_{113}, S'_{114}, S'_{115}\}$

---

[13] $MAX(S_{lk})$ means the node $S_{lki} \varepsilon S_{lk}$ such that for all $S_{lkn} \varepsilon S_{lk}$, $i \geq n$.

$u_1' = u_1 \cup <(S_{114}, S_{115})>$

$u_1' = <(S'_{111}, S'_{112}), (S'_{112}, S'_{113}), (S'_{113}, S'_{114}), (S'_{114}, S'_{115})>$

$b_\iota^1(i)$: $\forall j (j<i)$

For i=5, the above b-expression is satisfied. But for any value other than 5, $b_\iota^1(i)$=*false*. For values of i≥6 the node cannot be inserted since it violates the definition of data structures which requires a predecessor of the node to be present. In other words, for the given configuration, s(i)=6 is not an element of TCS_{11}, consequently $6 \notin CS_{11}$. Note that in the above insertion, since we are dealing with a 1-level structure we do not need to be concerned about the accessibility of the structure.

Another component of a characteristic set of operations is a deletion function.

Definition 3.3

A *primitive deletion function* (d-function) is a partial mapping: (following the same notation as in definition 3.2)

$$\delta : Z_L \times N^3 \rightarrow Z_L$$

where δ(z,l,k,m) is *defined* and equal to z'=(S', u'), if the following conditions are satisfied:[14]

1) $b_\delta^1(i)$ is satisfied with σ_{1k} with $s[i|S_{1km}]$,

2) If l<L then $\sigma_{1+1,q}$ satisfies $\alpha_{1+1}(S_{1+1,q,k})$ for some q,

---

[14] The deletion equation denotes: delete the node at level l of configuration z, indexed m, in node k of the $(l+1)^{st}$ level.

3) sucset(S_{lkm})=ϕ *AND* s_{lkm} contains either $S_{l-1,m,\emptyset}$ or $\Sigma_{undefined}$,

4) *if* S_{lkm}<MAX(S_{lk})

then- Let $S''=S-\{S_{lkm}\}\cup\{S_{lk\emptyset}\}$. Renumber the nodes at level l of S" starting with the structure contained in all the 1-successors of k and all the structures contained in nodes $S_{l+1,\kappa,q}$ for any κ such that q>k. The renumbering is done using the convention of definition 2.2, such that for every $S_{\lambda q}$, whose elements are renumbered to get $S''_{\lambda q}$, $(S_{\lambda q},u_{\lambda q})\simeq(S''_{\lambda q},u''_{\lambda q})$. Note that the renumbering would not affect the l-index(level indicator) of the nodes. Also $S_\lambda=S''_\lambda$ for $l<\lambda\leq L$ and $S_{\lambda\kappa}$ for $\lambda=l$ and $\kappa<k$. S' is then assigned to be the same as S".

else- $S'=S-\{S_{lkm}\}\cup\{S_{lk\emptyset}\}$.

The i-function described above restricts insertion only to those nodes which are not already in the structure. Likewise the deletion function may only delete the nodes with no successors. In certain cases where such restrictions are not acceptable we may use the i- and d-functions described in appendix D. In any case, as it will be demonstrated later on, one may define other functions using the primitive i- and d-functions.

A *probe function*, defined for a data structure Z_L, is any function whose codomain, for every configuration

$z=(S,u)\varepsilon Z_L$, is S. There are many different probe functions that one may define for a structure Z_L. The range of values of a probe function would always be a subset of the existing nodes in any given configuration of Z_L. In every configuration, $z\varepsilon Z_L$, for every level l, every structure (S_{lk},u_{lk}) is associated with the same set of probe functions defined for that level. Consider a **tree** data type, there are many different probe functions that one may define for it. For example a function that always returns the leftmost leaf node in a given configuration of the **tree** is a probe function; so is a function that given a node S_{lkm} returns one of its 1-successors. In order to be able to define such functions in general, we introduce the notion of the "primitive probe function." A primitive probe function may be considered to be a tool by means of which one may "correctly" define other probe functions.[15]

Given a configuration of a data structure and a node, S_{lki} in that structure, the primitive probe function returns a node, S_{lkj}, of that structure only if the latter is allowed access by the appropriate a-expression. In the **binary tree** data type, a possible probe function would be one that given a node as one of the arguments of this function, it would then return one of its 1-successors. Therefore with two primitive probe functions, one returning

---

[15] The correctness of a probe function is decided by whether the value of such function satisfies the a-expression for the structure or not. More will be said about this later in this chapter and chapter 5.

the left child and the other the right child, every **binary tree** configuration may be "scanned." Our choice of the primitive probe functions is arbitrary, and not every probe function does, or should, necessarily return the 1-successor of its argument. In a **stack** data type, say, not every 1-successor of every node is to be accessible. For instance, consider a configuration of depth 4 of the **stack**; if the probe function were to return the 1_successor of a node then the 1-successor of the node indexed 2 is 3. Since 3 is not the top node it is therefore not to be accessed. It is conjectured that using the following definitions of the primitive probe functions, one is able to define any other probe function such that all the nodes permitted for access may be probed.

Definition 3.4

Let z_L be a data structure, and $\alpha_l(i)$ be the a-expression associated with z_L at level l. A *primitive probe function* (p-function), at level $l \le L$, is a partial mapping:

$$\mathbf{p}_l : \quad z_L \times S \to S$$

such that for any $S_{lkm} \varepsilon S_{lk}$ and $z=(S_{lk},u_{lk})$ if

$\mathbf{p}_l(z,S_{lkm})=S_{lkn}$ then $S_{lkn} \varepsilon S_{lk}$ and

σ_{lk} satisfies $\alpha_l(i)$ with $s[i|S_{lkn}]$,

$\mathbf{p}_l(z,S_{lkm})=S_{lk\emptyset}$ otherwise.

It should be noted that not every node argument, S_{lkm} in the above p-function, causes the function to return a

value $S_{lkn} \varepsilon S_{lk}$, $n \neq \emptyset$. This was illustrated in the above **stack** example. In addition S_{lkm} itself is not necessarily in the range of the p-function. It may be any arbitrary node in S_{lk}. The only important issue is whether the value of the function, viz. S_{lkn}, satisfies the a-expression or not. If these conditions are not satisfied then the p-function evaluates to the empty node $S_{lk\emptyset}$.

Example

First consider the **list** and the **binary tree** data types. The p-expression (for the sake of brevity, assume a 1-level structure) for the **list** data type may be defined as: i<>i+1. To define a "p-function" for the **list** we employ Church's Lambda-notation (see appendix C), so that the p-function of the **list** is **p**(z,i)=λi. (i+1). This function takes a node, i, and a **list** configuration z as its arguments and returns a node labeled (i+1) corresponding to its 1-successor, if any, in z.[16] For the case of the **binary tree** data type; The corresponding p-functions are:

$$\text{p-functionL}(z,i)=\lambda i.\ \ i=\emptyset \rightarrow 1,\ 2i$$

$$\text{p-functionR}(z,i)=\lambda i.\ \ i=\emptyset \rightarrow 1,\ 2i+1$$

The a-expressions for both the **list** and **binary tree** are $i \geq 1$.

---

[16] Note that the p-function and the p-expressions are not necessarily related, as implied in this example. In majority of the cases, however, this approach enables one to use these p-functions recursively in order to enumerate the set of all nodes which are accessible. Also a node <lki> is represented by i, since l and k are assumed fixed.

Next we look at the p-function for the **stack** data type: This is defined recursively, so that it always yields the "top" node of the stack. Assume l and k are fixed, then

$p(z,i) = \lambda i.\ \forall j(i \geq j) \rightarrow i,\ p(z,i+1)$

Clearly the expression $\forall j(i \geq j)$ is satisfied in the presence of σ^r_{lk} if s(i) is equal to the "topmost" node in some S^r_{lk} in z. This expression is indeed the a-expression for the **stack** data type.

A node <l,k,m> of a data structure configuration, z, is *accessible* at level l iff $\langle l,k,m \rangle \varepsilon R^z_{lk}$, where R^z_{lk} is the set of nodes $\{i: \alpha_l(i)\}$ and $\alpha_l(i)$ is the a-expression associated with level l of z. For example, for every **stack** configuration (S_{lk}, u_{lk}), the set of accessible nodes is: $\{i: \forall j(j \leq i)\}$. Similarly for an **array** configuration, (S_{lk}, u_{lk}), $R^z_{lk} = \{i: i \geq 1\}$. It is needless to say that the predicates defining the latter two sets are the a-expressions for the **stack** and the **array** data types respectively.

The notion of accessibility of nodes is particularly useful in the area of maintaining the integrity and security of data in a data base environment. For example consider a set of data items shared by many users. In such an environment each user (or group of users) may have his own a-expression associated with the data in a manner that would confine him to his own allocated area(s). Such an

arrangement implies that the given set of data items would behave differently to each user.

Three types of primitive functions have now been identified. To characterize a data type these functions are put together to form a "characteristic operation set" of that data type.

Definition 3.5

A *Characteristic DSO Set* (CDSO), ψ, with respect to a data structure Z_L, is defined as a set:

$$\psi = I \cup D \cup P$$

where,

- I is a set of i-functions, i_l, one for each level $l \leq L$ of Z_L.
- D is a set of d-functions, d_l, one for each level $l \leq L$ of Z_L.
- P is a set of p-functions, p_l, one or more for each level $l \leq L$ of Z_L,

such that for every $z \neq (\{S_{L1\emptyset}\}, \flat)$, if $g_{l_r} \in I \cup D$ and

$$z = g_{l_n}(\ldots g_{l_2}(g_{l_1}(\phi, l_1, k_1, m_1), l_2, k_2, m_2) \ldots, l_n, k_n, m_n)$$

is defined then for every l and every k, the destruction set DS^z_{lk} is non-vacuous.

So far we have talked about three types of predicate expressions: p-, b- and a-expressions. These expressions may be associated with different levels of a data structure in the manner described earlier. We shall refer to these

predicate expressions, associated with each level l, as the *base predicate expressions(set)*, Π_l, associated with that level l of Z_L.

3.2- A Definition of Data Types

In chapter 2 the notion of data structure was introduced. In the preceding section we presented a set of operations that may be associated with a data structure. Using the elements of this set of operations one may make transitions from a configuration of the data structure to another. Recall that Z is the largest set of Σ-structures which is implementable wrt some family of p-expressions þ. We are often interested in only a subset of Z. That is only those configurations in Z that are "valid" configurations. For example if we are interested in list configurations of length less than 10 (say) then only a subset of the set of all lists is of interest. Let us now present a definition of data types.

Definition 3.6

Let Z be a data structure and ψ be a CDSO set defined wrt Z, then $t=(Z,\psi)$ is a *data type*.

Definition 3.7

Let $t=(Z_L,\psi)$, then a *configuration* of t is a pair (z,ψ) where $z \varepsilon Z_L$ is a data structure configuration and ψ is the set of DSO's.

Let us now introduce the concept of reachability of the elements of a data type.

Definition 3.8

Let z, $z' \varepsilon Z_L$. Let $G=I_U D$ denote the set containing i-functions, i, and d-functions, d, of ψ defined wrt Z_L. Then z' *is reachable from* z *wrt* ψ, denoted $z |-_{\psi} z'$, if for some $g \varepsilon G$ and l, k, $m \varepsilon$ **N**:[17]

1) z'=z *or*

2) z'=g(z,l,k,m), *or*

3) $z'' |-_{\psi} z$ and z'=g(z",l,k,m)

Definition 3.9

Let $t=(z,\psi)$ and $t'=(z',\psi) \varepsilon t$;[18] then the *reachability relation*(ρ-relation) on t is:

$$\rho=\{ (t,t'): z |-_{\psi} z' \}$$

In some cases where ψ is understood, we may say (z,z') is in the ρ-relation if $(t,t') \varepsilon \rho$.

Lemma 3.3

---

[17] z=z' means S=S' and u=u', where z=(S,u) and z'=(S',u').

[18] $t=(z,\psi) \varepsilon t$ denotes $z \varepsilon Z$, where $t=(Z,\psi)$.

Let $t=(Z,\psi)$ be a data type such that for $z \varepsilon Z$ and every l and k, σ_{lk} satisfies the b-expression for insertion, $b_l^1(i)$, with every $s(i)\varepsilon TCS_{lk}$, and σ_{lk} satisfies the a-expression, a_l, for every l and k, then z is reachable from ϕ wrt ψ.

Proof

In order to simplify the discussion, we shall only consider a single-level structure. The proof for the general case follows in a straight forward manner.

In chapter 2 it was shown that, every data structure Z is recursively enumerable. Using this fact we can show that every $z=(S,u)\varepsilon Z$, is reachable from ϕ wrt ψ. We employ an inductive argument on the number of nodes in the configuration $z=(S,u)$.

Let z^n denote the configuration $(S^n,\text{Þ})$ where n denotes the number of nodes in S^n other than the empty node $S_{11\emptyset}$. Then $z^{\emptyset}=(\{S_{11\emptyset}\},\text{Þ})$ is indeed reachable from ϕ by definition.

Let us now assume that for every configuration, z^{n-1}, with n-1 nodes, z^{n-1} is reachable from ϕ, then the question is if every $z^n=(S^{n-1}\cup\{S_{11j}\},\text{Þ})\varepsilon Z$ is reachable from ϕ. Let $z^n=(S^{n-1}\cup\{S_{11j},\text{Þ})\varepsilon Z$, and let us assume that $(S^{n-1},\text{Þ})=z^{n-1}$, then $i(z^{n-1},1,1,j)=z^n$. The latter equality is true, since S_{11j} must have satisfied the p-expression with some existing node of z^{n-1} otherwise $z^n \not\varepsilon Z$. This is of course a contradiction. Therefore $S_{11j}\varepsilon TCS$ and also $S_{11j}\varepsilon CS$ of z^{n-1},

recall that the b-expression is also satisfied. Hence by the definition of i-function $z^n = i(z^{n-1}, 1, 1, j)$. Therefore z^n is reachable from ϕ.

Using the above lemma, we can immediately say the following.

Corollary 3.2

Let $t=(Z_L,\psi)$; let $\phi|-_\psi z'=(S',p)\varepsilon Z_L$, if $b^1_l(S_{1km})$=true and $a_{l+1}(S_{l+1,q,k})$=true for some q and $l<L$, then $\phi|-_\psi z=(S' \cup \{S_{1km}\}, p)\varepsilon Z_L$.

Corollary 3.3

Let $t=(Z,\psi)$ be a data type; for every $z \varepsilon Z$ if $\phi|-_\psi z$ then $z|-_\psi \phi$.

Proof

The proof is a consequence of the definition of ψ.

Theorem 3.1

Let $t=(Z,\psi)$ be a data type. The reachability relation, ρ, defined on t is an equivalence relation.

Proof

The transitivity and reflexivity follows from the definition of ρ. The symmetry is a consequence of corollaries 3.2 and 3.3.

So far we have been examining a global view of data types. In the next section we shall delve into the properties possessed by each data type.

3.3- Data Types as Lattices

In section 3.2, it was shown that some elements of the structural component **Z**, of a data type t=(**Z**,ψ) may be either φ or i(z',l,k,m) where iεψ, z'ε**Z** and z' is reachable from φ. These elements of **Z** may be represented in terms of the elements of the corresponding "word algebra", also known as the *Herbrand Universe*. For example a **stack** of length 3 may be represented by its i-function, PUSH, namely: PUSH(PUSH(PUSH(φ,1),2),3). Similarly a **binary tree** configuration may be represented by i(i(i(φ,1),2),5).

Consider a data type **t**=(**Z**,ψ), then the *word algebra*, WORD(ψ)=(**W**,ψ), of **t** is defined as follows. For each constant in **Z** there exists a term f_0ε**W** of *rank* zero. Also for each non-negative integer there exists a term f_0ε**W** of rank 0. The elements of rank zero are called the *generators*, or *alphabet*, of the word algebra, WORD(ψ). For every i- and/or d-function f^4εψ, $f^4(x_1,x_2,x_3,x_4)$ there exists an element of **W** defined as any expression (also called *word* or *term*) of the form $f^4x_1x_2x_3x_4$ of rank m, where m is a positive integer, and there exists at least one x_i of rank m−1 and no x_i, 1≤i≤4, with rank greater than m−1. It

should be noted that for a data type $t=(Z,\psi)$, there exists only one constant, namely the starting configuration ϕ. The alphabet of the corresponding word algebra would then contain the generators corresponding to ϕ and the elements of the set of non-negative integers **N**. From now on, for the sake of clarity, we have taken the liberty of using the symbols ",", "(" and ")" in the representation of the terms of the word algebra.

Using the above representation, there are many terms of the word algebra, of a data type **t**, that represent the same configurations of **t**. Only one representation of such terms would be desired. Consider for example the term push(pop(push(ϕ,1),1),1) which represents the same configuration of a **stack** as push(ϕ,1) does. Under such circumstances we are only interested in a single representation. Let W_t represent the set of terms $W-R(N)$, where $t=(Z,\psi)$ and $word(\psi)=(W,\psi)$ and $R(N)$ is the set of elements of rank zero representing the elements of **N**. The set W_t contains one or more terms that correspond to some configuration $z\varepsilon Z$. Also in W_t there are certain terms that do not make any sense as far as t is concerned. In other words they do not represent any of the elements of **Z** that are reachable from ϕ. For example we may have a term in W_t, where t is a **stack** (say), in the form of

pop(pop(pop(ϕ,2),4),5).[19] Such a term does not of course make any sense in the **stack** data type.

As pointed out above there may be more than one way of representing a configuration of a data structure using the corresponding word algebra. In order to have a unique representation for every data structure configuration of a data type t, an equivalence relation is defined on W_t; this relation will be referred to as the "υ-relation." A υ-relation, υ, partitions W_t into equivalence classes resulting in a quotient set $W_t/_\upsilon$. The definition of υ on W_t will be presented shortly after we distinguish between the two kinds of terms that one may find in W_t, viz. either the "non error" kind or the "error" kind. The *non-error terms* are defined recursively as follows:

1) ϕ is a non-error term,
2) g(t l k m) is a non-error term if t is a non-error term and $\bar{g}(\bar{t},l,k,m)$ is reachable from ϕ, where $\bar{t}$ is the corresponding value of t in Z.[20]
3) There are no other non-error terms.

The set of all the non-error terms is referred to as NET. Clearly NET$\subseteq W_t$. The second category of terms in W_t, namely the set of terms t$\notin$NET, as noted earlier, are the error terms. The set of error terms in W_t, where $t=(Z_L,\psi)$,

---

[19] The l- and k-values for the node indices are assumed to be some fixed value, hence not specified.

[20] For a term g(t,l,k,m), $\bar{g}$(t,l,k,m) denotes the value of the function g.

may be generated as follows. Let g_{l_j} be either an i-function or a d-function in ψ, where $l_j \le L$. Then $t' = g_{l_j}(t,l,k,m)$ is an error term, and is referred to as an *overflow term* if at least one of the following is satisfied.

- g_{l_j} is an i-function and $\bar{t}$ is reachable from ϕ and $\bar{t}'$ is not reachable from $\bar{t}$.
- t is not an underflow term (see below) and not reachable from ϕ and g_{l_j} is an i-function.
- t is an overflow term.

Similarly, $t' = g_{l_j}(t,l,k,m)$ is an error term, denoted as an *underflow term*, if one of the following is satisfied.

- g_{l_j} is a d-function and $\bar{t}$ is reachable from ϕ and $\bar{t}'$ is not reachable from $\bar{t}$.
- t is not an overflow term and not reachable from ϕ and g_{l_j} is a d-function.
- t is an underflow term.

In summary, there are two types of error terms: the overflow and the underflow. The former set will be denoted as OFT and the latter will be referred to as UFT.

Let us now define the υ-relation. Let $t_1, t_2 \varepsilon W_t$; $t_1 \ \upsilon \ t_2$ iff either

1) Both t_1 and t_2 are non error terms and $\bar{t}_1 = \bar{t}_2$,.[21] or
2) Both t_1 and t_2 are overflow terms; or
3) Both t_1 and t_2 are underflow terms.

---

[21] $\bar{t}_1 = (S_1, u_1) = \bar{t}_2 = (S_2, u_2)$ means $S_1 = S_2$ and $u_1 = u_2$.

If $(t_1,t_2)\varepsilon\upsilon$, then this may be denoted as "t_1 is υ-equivalent to t_2." It is obvious that the υ-relation is an equivalence relation. Hence W_t can be partitioned into equivalence classes. To choose a representative for each class and to be consistent in our approach, we select a term of that class known as the "minimal form" of the terms in that class. A minimal form(MF) of a term, t, representing a configuration, z, is a term t_{MF} which is free of redundant deletion or insertion function symbols. We shall shortly demonstrate that for every data type the underlying system is "Church-Rosser" [CHU 36, ROS 70, SET 74]. Hence every non-error term may be condensed to a unique MF.

In order to have a formal notion of MF, and indeed to show the existence of such a unique representation for each configuration, we adopt the following approach. A *replacement system*, $(W,\Rightarrow)$, is a set of objects and a transformation on the objects[SET 74]. Let us assume that, for our purpose, W is the set W_t, where $t=(Z_L,\psi)$. Define $\Rightarrow$ as a transformation on W_t as described below. Let t, t_1, $t_2\varepsilon W_t$, and g_l denote either an i-function or a d-function symbol, then t_1 *reduces* to t_2, or $t_1\Rightarrow t_2$, if at least one of the following rules is satisfied.

Rule 1:

$$t_1=d_l(g_{l_n}(..g_{l_1}(i_l(t,l,k,m),l_1,k_1,m_1)..),l_n,k_n,m_n),l,k,m)$$

and

$$t_2=g_{l_n}(..g_{l_1}(t,l_1,k_1,m_1)...),l_n,k_n,m_n) \textit{ AND } t_1,\ t_2\varepsilon NET.$$

Or **Rule 2:**

$t_1=i_{l_n}(\ldots i_{l_2}(i_{l_1}(t,l_1,k_1,m_1),l_2,k_2,m_2)\ldots,l_n,k_n,m_n)\varepsilon NET$ where $t\varepsilon NET$ and there are no instances of a d-function symbol in t_1, and

$t_2=i_{l_n}(\ldots i_{l_1}(i_{l_2}(t,l_2,k_2,m_2),l_1,k_1,m_1)\ldots l_n,k_n,m_n)\varepsilon NET$ such that at least one of the following conditions is satisfied:

i) $l_2>l_1$

ii) $l_2=l_1$ and $k_2<k_1$

iii) $l_2=l_1$ and $k_2=k_1$ and $m_2<m_1$.

Or **Rule 3:**

$t_1\varepsilon OFT$ (UFT) and $t_2=oft(uft)$, where oft(uft) is any one of the terms in OFT(UFT) which may be selected arbitrarily.

The following theorem shows that when a non error term t_1 reduces to a term t_2, the value of the term does not change, i.e. if $t_1=>t_2$ then $\bar{t}_1=\bar{t}_2$.

Theorem 3.2

Let $t=(Z_L,\psi)$ be a data type, and let t_1, $t_2\varepsilon W_t$ be non error terms; if $t_1=>t_2$ then $\bar{t}_1=\bar{t}_2$.

Proof

There are four ways of performing a reduction. These are considered separately below.

i) let g_l denote either an i-function or a d-function at level l; and let

$t_1=d_l(g_{l_n}(\ldots g_{l_1}(i_l(t,l,k,m),l_1,k_1,m_1)\ldots),l_n,k_n,m_n),l,k,m)$

and

$t_2=g_{l_n}(\ldots g_{l_1}(t,l_1,k_1,m_1)\ldots),l_n,k_n,m_n)$

Let us now assume that the set of nodes inserted in the configuration represented by t_1 is S_i, and let the set of nodes that are deleted in t_1 be S_d. Then $\bar{t}_1=(S_i-S_d,\mathfrak{p})$, where $\mathfrak{p}$ is the family of p-expressions associated with Z_L. Similarly, the value of t_2 is

$\bar{t}_2=(S_i-\{S_{lkm}\}-(S_d-\{S_{lkm}\}),\mathfrak{p})=(S_i-S_d,\mathfrak{p})=\bar{t}_1$

ii) Let $t_1=i_{l_n}(\ldots i_{l_2}(i_{l_1}(t,l_1,k_1,m_1),l_2,k_2,m_2)\ldots l_n,k_n,m_n)$ where there are no occurrences of a d-function symbol in t_1. If $l_2>l_1$ and

$t_2=i_{l_n}(\ldots i_{l_1}(i_{l_2}(t,l_2,k_2,m_2),l_1,k_1,m_1)\ldots l_n,k_n,m_n)$

then we want to show that $\bar{t}_1=\bar{t}_2$. Two cases are considered:

a) S_{l_2,k_2,m_2} contains a structure in which S_{l_1,k_1,m_1} occurs. This cannot possibly be the case, since it implies insertion of a node without the presence of its supporting node.

b) S_{l_2,k_2,m_2} does not contain a structure in which S_{l_1,k_1,m_1} occurs. In this case the two insertions are independent in the sense that if the higher level node is inserted first, with the insertion of the lower level node next, it cannot possibly alter the value of the term t_1.

iii) Let t_1 and t_2 be as in case (ii) above and $l_2=l_1$, Commuting i_{l_1} and i_{l_2} does not alter $\bar{t}_1$ because of the following. The node S_{l_2,k_2,m_2} is inserted after S_{l_1,k_1,m_1} in

$\bar{t}_1$ where $k_2<k_1$, the latter implies that $m_2<m_1$. Consequently if $m_2<m_1$ and S_{l_2,k_2,m_2} can be inserted after S_{l_1,k_1,m_1} then it can surely be inserted before it without changing $\bar{t}_1$.

iv) Finally, consider the case where $l_1=l_2$ and $k_1=k_2$ which is similar to that of case (iii) above.

We can also think of => as a binary relation on *W*, i.e. $t_1 \Rightarrow t_2$ may also be expressed as $(t_1,t_2)\varepsilon\Rightarrow$. Let $\Rightarrow^0$ denote the identity relation, and let $\Rightarrow^i = \Rightarrow . \Rightarrow^{i-1}$ for all i>0, where "." denotes composition of the two relations => and $\Rightarrow^{i-1}$, i.e. if x=>y and $y \Rightarrow^{i-1} z$ then $x \Rightarrow^i z$. The *reflexive closure* of =>, denoted by $\Rightarrow^{\#}$, is given by $\Rightarrow^{\#} = \Rightarrow^0 \cup \Rightarrow$. The *reflexive transitive closure* of =>, denoted by $\Rightarrow^*$, is given by $\Rightarrow^* = \Rightarrow^{\#} \cup \Rightarrow^1 \cup \Rightarrow^2 \cup \ldots$. An element $t\varepsilon W_t$ is *irreducible* or in *minimal form*(MF) under => if there is no t' such that t=>t'. The *completion* of =>, denoted by $\Rightarrow^{\surd}$ is $\{(x,y) : x \Rightarrow^* y$ and y is irreducible $\}$, and if $(x,y)\varepsilon\Rightarrow^{\surd}$ then y is the MF of x. A replacement system, (*W*,=>) is *finite* if for all $x\varepsilon W$, there is a constant k_x such that if $x \Rightarrow^i y$ then $i<k_x$ [SET 74]. Finally, a finite replacement system is Finite Church-Rosser(FCR) if $\Rightarrow^{\surd}$ is a function. Note that the latter implies a unique MF for every element.

Theorem 3.2

Let **t**=(**Z**, ψ) be a data type and $(W_t, \Rightarrow)$ be the corresponding replacement system. $(W_t, \Rightarrow)$ is Finite Church-Rosser(FCR).

Proof

To show (W_t, =>) is FCR two conditions must be satisfied(see Theorem 2.1 in [SET 74]):

a) (W_t , =>) is finite, and

b) for all z_1,z_2,w,x in W_t, if z_1 and z_2 are equivalent and z_1=>w and z_2=>$^{\#}$x imply that for some y and z, where y is equivalent to z, w=>*y, and x=>*z. This is shown pictorially figure 3.1.

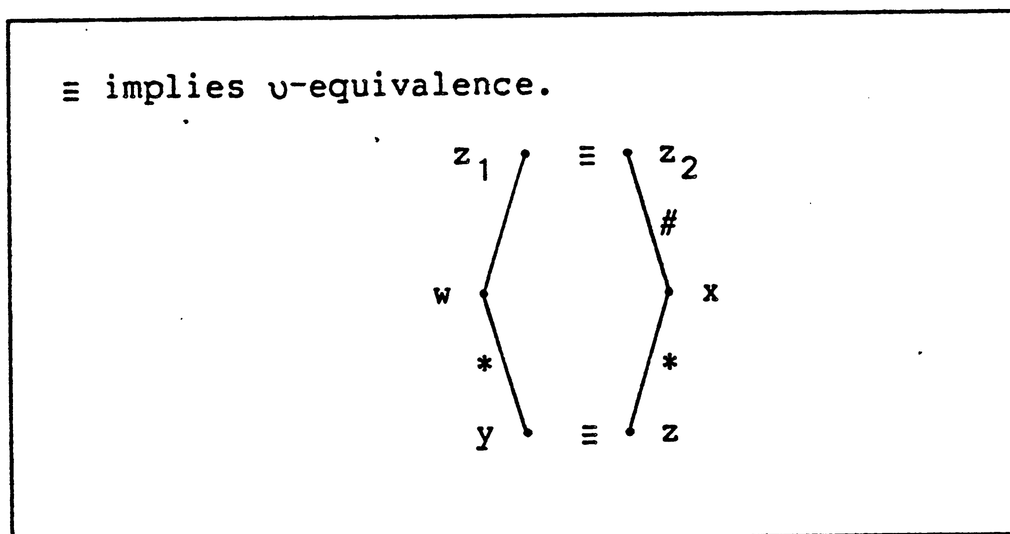


Figure 3.1.
Reachability of Elements in Z.

To show that the above condition (a) is satisfied, consider the 4 types of reduction that may be performed on a term $t \varepsilon W_t$. The number of times that the reduction rule 1 may be applied on t is bounded by the number of d-function symbols in t. The number of times that the reduction rule

2(i) may be applied on t is also bounded by a constant multiple of the number of i-function symbols in t less the number of d-function symbols. And indeed the remaining two types of reduction are bounded by even a smaller number than that of 2(i). Hence for any given term t the total number of reductions is bounded.

To show the satisfaction of condition (b) consider $z_1, z_2 \varepsilon W_t$ such that z_1 is υ-equivalent to z_2. Therefore if $\bar{z}_1=(S_1,u_1)$ and $\bar{z}_2=(S_2,u_2)$, then $(S_1,u_1)=(S_2,u_2)$. If $z_1 \Rightarrow w$, where $w=(S_w,u_w)$, then $(S_w,u_w)=(S_1,u_1)$. This is true since the => transformation does not affect the underlying structure represented by z_1, namely its value. By the same argument, $\bar{y}=(S_y,u_y)=(S_w,u_w)=(S_1,u_1)$ and $\bar{z}=(S_z,u_z)=(S_x,u_x)=(S_2,u_2)$. Therefore $(S_y,u_y)=(S_z,u_z)$, i.e. y and z are υ-equivalent.

As a result of the above theorem, we conclude that, there exists a unique minimal form that every term of a data type may be reduced to. Clearly all the terms reducing to the same MF are members of the same equivalence class of υ. For example a **stack** configuration of length 3 may be represented by any of the following formulas, where i denotes the push function and d denotes the pop function,

$d(i(i(i(d(i(i(\phi)))))))$

$i(i(d(i(i(\phi)))))$

$i(i(i(\phi)))$

The latter form represents the MF, and would be considered as the representative of its own υ-equivalence class.

Finally let W_θ denote an epimorphic image of $NET_U UFT_U OFT$ containing all the MF's of the elements of W_t; such that W_θ contains exactly one element, the MF, corresponding to each equivalence class of $W_{t/\upsilon}$. Thus W_θ contains oft, which is the image of all the elements of OFT, and exactly one element, uft, which is the image of all the elements of UFT; and all the MF's of the elements of NET. Note that this mapping is no more than the completion of =>; i.e.

$$W_\theta=\{y : x=>^{\checkmark}y \ \& \ x\varepsilon W_t \}$$

With the above set of terms, W_θ, we can define a *word algebra of minimal forms*, namely (W_θ,ψ), as follows. Let t_{MF} denote the MF of $t\varepsilon W_t$, then if $t_{MF}\varepsilon W_\theta$ and g is either an i- or a d-function in ψ then $g(t_{MF},l,k,m)=t'_{MF}$,[22] where $t'=g(t_{MF},l,k,m)\varepsilon W_t$. We may now show that for every element $z\varepsilon Z$, of a data type $t=(Z,\psi)$, one can find what its corresponding minimal form representation is.

Lemma 3.4

Let $t=(Z,\psi)$; for every $z=(S,\flat)\varepsilon Z$ we can decide what its corresponding MF representation, $t\varepsilon W_\theta$, is.

---

[22] Note that we have used the same function symbols for both (W_t,ψ) and (W_θ,ψ); the context should remove any ambiguity.

Proof

Let $z=(S,þ)$ and let t_{MF} denote the MF of t. Construct t_{MF} as follows: if $S=\{S_{1k\emptyset}\}$ then $t_{MF}=\phi$ else starting with the nodes $S_{lkm} \varepsilon S$ with the largest first index l in S, write τ, where

$$\tau=i(\ldots i(i(\phi,l,k_1,m_1),l,k_2,m_2),\ldots,l,k_n,m_n)$$

such that $k_1 \le k_2 \le \ldots \le k_n$ and $m_1 \le m_2 \le m_n$.(In case of ambiguity the k-index has the higher precedence.) Next select the nodes $S_{l-1,k',m'} \varepsilon S$, if any, and write τ' as follows:

$$\tau'=i(\ldots i(\tau,l-1,k_1',m_1'),\ldots,l-1,k_n',m_n')$$

Repeat the above procedure until all the nodes in S are exhausted. The resulting term, t, is in MF since the above node selection process is no more than the application of rule 2 of reduction discussed earlier.

Test to see if $\bar{t}$ is reachable from ϕ wrt ψ. It should be noted that if the insertion is made in the sequence of nodes in t, then $\bar{t}$ is not necessarily reachable from ϕ in that sequence. Thus testing for reachability must be such that if there is a "valid" way to get to $\bar{t}$ then $\bar{t}$ is considered reachable irrespective of the sequence of nodes in its corresponding MF. In general if t is given, form the set $S=\{S_{lki}:\langle l,k,i\rangle$ occurs in $t\}$. By lemma 2.2 we can decide if $(S,þ) \varepsilon Z$ or not, if it is not in Z then t_{MF}=oft, else proceed as follows. For every intermediate term τ, τ', τ'',... and finally t we can perform the reachability test as prescribed below.

test τ to see if $\bar{\tau}$ is reachable from ϕ, if so

test τ' to see if $\bar{\tau}'$ is reachable from ϕ, and so on At each stage to see if $\bar{\tau}^n$ is reachable from $\bar{\tau}^{n-1}$, test for every l and k values, that occur in τ^n but not in τ^{n-1}, all the possible sequences of insertion of the node indices for a given l and k. If for at least one sequence $\bar{\tau}^n$ is reachable from $\bar{\tau}^{n-1}$ then $\tau^n_{MF} \neq$ oft, i.e. τ^n_{MF} may be obtained by the above procedure.

Finally the fact that t is unique follows from theorem 3.3.

<u>Theorem 3.4</u>

υ is a congruence relation on (W_t, ψ).

<u>Proof</u>

υ is obviously an equivalence relation. To show that υ is also a congruence relation it must be demonstrated that it exhibits the substitution property. Assume $t \upsilon t'$, i.e. t and t' are υ-equivalent; then the claim is that

$$\tau = g(t,l,k,m) \ \upsilon \ \tau' = g(t',l,k,m)$$

where g is either an i-function or a d-function in ψ. Now, since $t \upsilon t'$ then $\bar{t} = \bar{t}'$, therefore

$$\bar{\tau} = \bar{g}(\bar{t},l,k,m) = \bar{g}(\bar{t}',l,k,m) = \bar{\tau}'$$

But if $\bar{\tau} = \bar{\tau}'$ then $\tau \upsilon \tau'$, i.e.

$$g(t,l,k,m) \ \upsilon \ g(t',l,k,m)$$

Note that if τ is an error term then τ' would also be an error term of the same nature, i.e. they both would be

either overflow terms or underflow terms. As a result τ would still be υ-equivalent to τ'.

∎

Notation- In the following discussions i^*z_2 implies an application of a sequence of length zero or more of the same or different i-functions. Similarly in d^*z_1, d^* denotes zero or more applications of d-functions. Also $i^{\#}$ ($d^{\#}$ denotes a sequence of at least one i- (d-) function. Such sequences of all i-functions (d-functions) are referred to as *i-monotonic*(d-*monotonic*). A *bitonic* sequence is obtained as follows. A monotonic sequence is bitonic; if β_1 and β_2 are bitonic then $\beta=\beta_1\beta_2$ is also a bitonic sequence. The node indices <l,k,m> may be dropped for notational brevity in places where the ambiguity is not important. Thus, $i(z,l,k,m)$ is denoted by $i(z)$ or iz. Finally we may use $(i^k)_j$ to denote a sequence of, possibly distinct, i-functions of length k.

<u>Lemma 3.5</u>

Let $t=(Z,\psi)$ be a data type. Every term $t \varepsilon NET$ is υ-equivalent to a term $i^*\phi \varepsilon W_\theta$.

<u>Proof</u>

The set W_θ contains the MF of every term $t\varepsilon NET$. Consider a term $t\varepsilon NET$, t cannot be $d^{\#}\phi$. If so then $t=d^*d\phi$. Let us consider the value of $t'=d\phi$, namely $\bar{t}'$. According to

the definition of d-function, in order to have a value for $d\phi$, the node to be deleted must be in the DS of the configuration, in this case, ϕ. But the DS of the ϕ is empty since the TDS of ϕ is empty since there are no nodes in ϕ other than the empty node, $S_{1k\emptyset}$. Hence $d\phi$ is an element of UFT, thus $t \notin$NET.

Now, if t is an i-monotonic sequence concatenated with ϕ, then, by rule 2 of reduction, t may be reduced to its MF in the form of $i^*\phi \varepsilon W_\theta$. If t is not a monotonic sequence, by definition of =>, for every deletion of a node there must have been an insertion of that same node. This is a necessary condition since for a node to be deleted it must be an element of TDS. The latter implies that the node must have been inserted into the configuration otherwise deleting a non-existent node would result in $t\varepsilon$UFT(OFT). The latter is of course contrary to our original assumption that $t\varepsilon$NET. Thus by multiple applications of =>, using the first rule of reduction, all instances of the d-function symbols may be eliminated resulting in an i-monotonic sequence concatenated with ϕ. This would then reduce the problem to the one we considered earlier.

Every element $t\varepsilon W_\theta$-{uft,oft} corresponds to a configuration $z\varepsilon Z$ which is reachable from ϕ. The configuration $z=\bar{t}$ is the value of t. In order to define values for the two terms uft and oftεW_θ we introduce two

special elements, namely *top* ($\top$) and *bottom* ($\bot$), associated with every data type t. These two elements will be referred to as the *improper elements* of t (or Z). Note that they are not necessarily elements of Z. The improper elements are intended to "absorb" all the error instances of that data type. We may think of $\top$ and $\bot$ as representatives of all the elements that may, or may not, be in Z and not reachable from ϕ. The value of oft (uft) is defined as the $\top$ ($\bot$). Note that the value of a term $t \varepsilon W_t$, where t is υ-equivalent to oft (uft) is also defined as $\top$ ($\bot$).

Intuitively, the uft and the oft elements are really a "trap" as defined by [AFS 80]. It should be noted that $\top$ and $\bot$ are not reachable from ϕ or any configuration which is reachable from ϕ, whereas oft and uft are "monotonically reachable" (see below) from every $t \varepsilon W_\theta$.

Let us now examine some of the properties of the word algebra of a data type t. We define a *monotonic reachability relation*(MRR), >>, on W_t as follows. Let $t_1, t_2 \varepsilon W_t$, $t_1 >> t_2$, read t_1 is monotonically reachable from t_2, if $t_1 = i^* t_2$ ($t_1 = d^* t_2$). Note that the monotonic reachability relation is not symmetric but it is both reflexive and transitive. Henceforth, unless otherwise specified, we shall only deal with the cases where $t_1 >> t_2$ implies $t_1 = i^* t_2$, where i^* is an i-monotonic sequence. Above definition defines MRR on W_t; we can also define the MRR,

>>, on the corresponding set W_θ in a similar manner.[23] Let t_1, $t_2 \varepsilon W_\theta$, then $t_1 >> t_2$ if

$$t_1 = i^* t_2 \text{ OR } \exists t \varepsilon W_t \ (t \ \upsilon \ t_1 \ \& \ t = i^* t_2 \text{ OR } t \ \upsilon \ t_2 \ \& \ t_1 = i^* t)$$

Lemma 3.5

Let $t = (Z_L, \psi)$ be a data type. Then the Monotonic Reachability Relation, denoted by >>, is a partial ordering on W_θ.

Proof

To show >> is a partial ordering, it suffices to demonstrate that it is transitive, reflexive and antisymmetric. Let t_1, $t_2 \varepsilon W_\theta$. The transitivity and reflexivity of >> is obvious from the definition of >>. The relation >> is also antisymmetric since if $t_1 >> t_2$ then for some sequence i_1^*, $i_1^* t_2 = t_1$. If $t_2 >> t_1$ then $t_2 = i_2^* t_1$ for some i-monotonic sequence i_2^*. The only possible way for $t_1 = i_1^* t_2$ and $t_2 = i_2^* t_1$ to be true is if $t_2 = i^0 t_1$, therefore $t_2 = t_1$.

Theorem 3.5

For a data type $t = (Z_L, \psi)$, $(W_\theta, >>)$ is a lattice.

Proof

First consider a single-level structure. We have already shown that $(W_\theta, >>)$ is a po. To show that it is a

[23] Note that we are using the same symbol >> in both cases of W_t and W_θ to define MRR on these sets. The context removes any ambiguity.

lattice we just need to demonstrate that for every pair of elements t_1, $t_2 \varepsilon W_\theta$ there exists a unique lub and a unique glb [STO 77].

Let $t=i(\ldots (i(i(\phi,m_1),m_2) \ldots m_M) \varepsilon W_\theta$, then its corresponding structure, if any, is defined as (S,Þ) where $S=\{S_j$: j is a node index occurring in t}. Let t_1, $t_2 \varepsilon NET$, define $lub(t_1,t_2)=t_3$ where t_3 is the MF of the term representing $(S_1 \cup S_2$, Þ). Recall that by lemma 3.4 it can be decided what the MF representation of z=(S,Þ)εZ is. Define $glb(t_1,t_2)=t_3$, where t_3 is the MF of the term representing $(S_1 \wedge S_2$, Þ). If t_1 and/or t_2 are error terms, then define

$$lub(t_1,oft)=oft$$
$$lub(t_1,uft)=t_1$$
$$glb(t_1,oft)=t_1$$
$$glb(t_1,uft)=uft$$

If $lub(t_1,t_2)=t_3$ then t_3 is unique. That is if there exists a $t>>t_1$ and $t>>t_2$ then it must be the case that $t>>t_3$. To show this, let us assume that such a term exists, i.e. $t_3>>t$ and $t>>t_1$ and $t>>t_2$. If this is the case then t corresponds to a structure (S,Þ) as defined above. Since $t>>t_1$ and $t>>t_2$ then either S must at least contain both S_1 and S_2; i.e. the least value of $S=S_1 \cup S_2$ which is indeed the same as S_3 defined above or t is oft. The former case implies that $t=t_3$. The latter, i.e. t=oft, is true only if t_3=oft, hence t_3 is unique.

The case for $glb(t_1,t_2)$ is very similar to the above. That is if $glb(t_1,t_2)=t_3$, then there exists no other term t such that $t_1 >> t$ and $t_2 >> t$ and $t >> t_3$. Employing the same approach as above, assume $t >> t_3$, then S, the set of nodes corresponding to t, can be at most $S_1 \wedge S_2$, which is exactly the same as S_3. Hence $t=t_3$.

Finally for the general case of L-level data structures the proof is simply a straight forward extension of the above.

The CDSO set, defined earlier, requires the presence of both the i- and d-functions, as well as the probe functions, for every level of the data type. However in order to generate all the possible configurations of a data type $t=(Z_L,\psi)$, not every element of ψ is necessarily needed. One only needs the "generators" of t which are defined below.

Definition 3.10

Let $t=(Z,\psi)$ where $\psi=I_U D_U P$. A *generator function* (or a *constructor*) of t is an i-function ι (a d-function, δ) such that if $z \varepsilon Z$ and z is reachable from ϕ wrt ψ, then z is not reachable from $\phi \varepsilon Z$ wrt $\psi-\{\iota\}$ ($\psi-\{\delta\}$).

Note that for those data types that we have been dealing with so far, i.e. with no primitive nodes, the only generators are the elements of $I \subseteq \psi$. Clearly, in the stack,

push is a generator function; whereas the function pop is not since there exists no d-monotonic sequence in the word algebra of MF's of the **stack**, i.e. the starting configuration followed by a sequence of pop's.

The following results may be readily concluded using the latter theorem 3.4.

Corollary 3.4

Let $t=(Z,\psi)$ be a data type, and let (W_θ,ψ) be the corresponding word algebra of MF's, then uft is the least fixpoint [STO 77] for every d-function $d\varepsilon\psi$; and oft is the greatest fixpoint for every i-function $i\varepsilon\psi$.

The above corollary implies that the i- and d-functions in (W_θ,ψ) are "strict" functions [STO 77]. Thus if a term t is an element of OFT (UFT) then no matter what sequence of i- or d-functions are applied to t the result is always an element of OFT (UFT).

We can reassert the fact that every configuration of $z\varepsilon Z$, which may be represented by an element of NET, is monotonically reachable from ϕ. This is demonstrated below.

Corollary 3.5

Let $t=(Z_L,\psi)$, then

$$glb\{t:\ t\varepsilon W_\theta \wedge t\neq uft\}=\phi$$

Corollary 3.6

Let $t=(Z_L,\psi)$ be a data type and let $X=(W_\theta,>>)$ be the corresponding lattice. Then for every configuration, if there exists n nodes at level $l \leq L$ which contain $S_{l-1,k,\emptyset}$, then that configuration is the glb of n sets of terms of W_θ, say W_i, where each $(W_i,>>)$ is a sub-lattice of X and isomorphic to the lattice of the structure defined at level $l-1$.

Intuitively, the above corollary states that, in every node, S_{lki}, we can construct all the possible configurations of the structure defined at the next lower level(s).

3.4- The Primitive Data Types

The data structure operations, defined earlier, explicitly referred to levels one and above. Thus level zero structures, to which we referred to as primitives, could not be subjected to any insertion, or deletion, operation defined earlier. In this section it is intended to justify this exception and formalize the concept of what is commonly referred to as the "primitive (data) types."

There are a number of reasons that has made it apparently desirable to treat primitive data types differently from any other ("non-primitive") data type. Firstly almost all the conventionally accepted, primitive data types such as the **integer**, **real**, **char** etc. have been

taken as already well established (defined) by the system (compilers). Secondly, primitive data types are assumed to be well understood concepts, for instance the **integer** and the **real** data types. In most cases there exists a mathematical model to describe such data types. This is exemplified by the Peano's axioms or the theory of real numbers.

But both of the above reasons are not, in our view, valid arguments. The former point raised the issue of primitive data types being "built-in" data types. However it is not at all uncommon to have a machine with a built-in **stack**! Does this imply that the **stack** is a primitive data type? The widespread use of the **stack** data type as a built-in (hardware) data type strongly raises the importance of implementing many more non-primitive data types in a likewise manner. As for the second issue, raised above, although most commonly used primitive types are well understood concepts there may be other primitive types that are not so well understood. For example, we may want to specify a subset of the **integer**; or describe the **days_of_week** data type. Such concepts are presently being employed in some high level languages such as Pascal as the "user-defined" data types.

Most, if not all, authors employing graphs have not presented specification schemes that are equally applicable to primitive types [MAJ 77B, EAR 73, SHN 74]. In contrast

the advocates of the algebraic approach do not make any distinction between primitive data types and other non-primitive data types. However, their treatment of parameterized data types does imply an implicit presence of the primitives as "parameter types"[GHM 76, TWW 79, GOG 78]. In their approach a data type such as the **stack** may not be defined independently of a "primitive" or a parameter data type. Hence the **stack** is defined as **stack-of-( )**. As a result a great deal of work has been unnecessarily added to formalize the parameterization of data types. In our model, however, this is not the case. All data types may be specified independently of other data types as prescribed by the model. Hence parameterization is just another "type manipulation operation"(see the next chapter).

Informally, a primitive data type is defined as one which is "transparent" to the user. In a **stack-of-integers**, the **stack** represents the organization of the **integer**, i.e. the discipline under which they are maintained, whereas the integers are concrete objects both written into and read from the **stack** by the user. By the same arguments, in a **list-of-char**, the **char** constitutes the primitive data type. Because of the indivisibility of the primitive data types, we have designated level Ø of Σ-structures as primitive nodes. This approach enables us to define data types irrespective of their "parameter" data type. The latter can always be embedded in the former with a type manipulation operation.

3.4.1- Operations to Characterize Primitive Data Types

A primitive data type is a data type whose structural component is always of level zero. As in any other data type it needs a set of operations to characterize its behaviour. The **integer**, for example, may be characterized by a characteristic set of operations, viz. its "i-function" and "d-function", *suc* and *pred*. The probe function is no longer of necessity since at level zero the structures are indivisible.

A characteristic data structure operation(CDSO) set for a primitive data type is composed of only the insertion functions and the deletion functions. An insertion function would cause a transition from a given configuration to another. A deletion function would do the inverse of an i-function. The correctness of these functions is ensured by their boundary conditions which are maintained by their appropriate b-expressions(see later). Consider the **string** data type with 26 "constructors", namely the 26 letters of the alphabet. Each application of any of these i-functions results in a configuration which may be represented by the concatenation of that particular letter of the alphabet to the current configuration. For instance, starting from the null string ϕ, i_1 applied to the null configuration results in "A" (say), which is represented as $i_1(\phi)$. Moreover $i_{22}(i_1(\phi))$=VA, and $i_{13}(i_{15}(i_{22}(i_1(\phi))))$=NOVA.

We have already pointed out that a necessary condition for a characteristic set of data structure operations is to ensure that any two configurations are reachable from each other wrt this set. Now consider the **bool** data type with the two i-functions i_1 and i_2, where $i_1\phi$ represents FALSE and $i_2\phi$ represents TRUE. These two i-functions, on their own, do not yield a complete DSO set, since $i_1(\phi)$ is not reachable from $i_2(\phi)$ and vice versa. An obvious solution would be the presence of the "NOT" function. However in order to establish a consistent approach, and indeed a more general one, we require a deletion function for each i-function of **bool**, namely d_1 and d_2, such that

$$d_1(i_1(b))=b \quad \text{and} \quad d_2(i_2(b))=b$$

As a result *false* is now reachable from *true* and vice versa.

A characteristic DSO set for a primitive data type is composed of two types of "primitive functions": the i-function and the d-function. If $t=(Z_\emptyset, \psi)$ is a primitive data type, its i-function i is a mapping:

$$i: \quad Z_\emptyset \rightarrow Z_\emptyset$$

similarly, the d-function

$$d: \quad Z_\emptyset \rightarrow Z_\emptyset$$

A d-function causes the inverse of an i-function. It "undoes" what its corresponding i-function "does." As a result if $i_2(i_1(i_{12}(i_{12}(\phi))))$=BALL then we get the equation: $d_2(i_2(i_1(i_{12}(i_{12}(\phi)))))=\text{ALL}=i_1(i_{12}(i_{12}(\phi)))$.

Definition 3.11

A *primitive data type* is a pair $(Z_\emptyset, \psi)$, where $Z_\emptyset$ is a $\emptyset$-level Σ-structure and ψ is a set of partial functions defined on $Z_\emptyset$ such that for $o \varepsilon \psi$

$$o: \; Z_\emptyset \rightarrow Z_\emptyset$$

In chapter 2 we introduced a numbering scheme for the nodes of a Σ-structure in which each node could be uniquely identified (cite definition 2.2). Thus a node $\langle l, k, m \rangle$ describes a node at level l, contained in the kth node of level l+1, and m indicates its index. For the primitive nodes however the latter is not of any value since primitive data structure configurations are indivisible. Therefore a different meaning is attached to m. For a primitive data type, m designates a numbering of the data structure configurations of that data type. More simply, each integer m may represent a unique configuration of the primitive data type. For example the ASCII code for string symbols indicate the value of m for each individual letter of the alphabet, i.e. 41 represents "A" and 42 represents "B", and so on.

Each configuration $z \varepsilon Z_0$ is merely an indivisible element rather than a pair (S,u) as in the case of the non-primitve data types. Furthermore, the p-expressions are of no more value in the case of the primitive data types. With the above convention of indexing for a data structure

configuration terminated with primitive nodes, each node S_{1qk} contains at most one element $S_{\emptyset km}$ where the index m designates a configuration of the primitive data type. The insertion and deletion operations at level $\emptyset$ would of course differ from that of any other level described earlier. This is discussed below.

3.4.2- Constructors of Primitive Data Types

Before delving into the definition of the i- and d-functions, let us introduce the b-expressions for the case of the primitive data types. The b-expression for primitive data types perform basically the same function as in the case of the non-primitive data types. However there are some minor differences between the two types of expression. The bwff's for primitive data types contain exactly one free variable as before. For primitive data types, however, the free variable denotes an element $z \varepsilon Z_0$ rather than a node index. Every i-function has a b-expression associated with it, so does every d-function. There may exist more than one b-expression, one for each i-function and one for each d-function. The function of a b-expression, associated with an i-function(d-function) is basically to determine for what elements $z \varepsilon Z_\emptyset$ is the i-function(d-function) defined. For example, if $b_i(z)$, the b-expression for an insertion function i, is satisfied then $i(z) \varepsilon Z_\emptyset$. Similarly if $b_d(z)$, the b-expression for a d-function d, is satisfied then $d(z) \varepsilon Z_\emptyset$. For example, if d denotes the *pred* function for

the **integer**, the b-expression for d, denoted b_d, has only one free variable and is defined as:

$$b_d(z) = \text{true} \quad \text{if } z \neq \phi$$
$$= \text{false} \quad \text{otherwise}$$

which implies that the predecessor function may be applied to any integer (the configurations of the **integer**), including zero, but not to the starting configuration, ϕ. On the other hand, the i-function to represent the integer 0, $i_\emptyset$ (say), has a b-expression defined as:

$$b_{i_\emptyset}(z) = \text{true} \quad \text{if } z = \phi$$
$$= \text{false} \quad \text{otherwise}$$

One may use the d-function, mentioned above, to reach the integer "-2"(say) starting from "0", i.e. $d(d(i_\emptyset(\phi)))$.

The definition of the structure in which the satisfaction of a b-expression, for primitive data types, may be determined, is slightly different from that described earlier; this distinction is described next. Let $Z_\emptyset$ be a set of 0-level configurations. A representation for the elements of this set may be found in its corresponding word algebra. The set $Z_\emptyset$ is simply considered to be the universe of the structure. All the variables, in a b-expression for a primitive data type, denote the elements of $Z_\emptyset$ and all the constant symbols are also assigned to the elements of $Z_\emptyset$. Finally, any computable function and decidable relation may be defined in the structure as before.

We are now in a position to present a more specific definition of the constructors of a primitive data type, namely its i- and d- functions.

Definition 3.12

A *primitive insertion function*, or an i-function, ι, for a primitive data type $t=(Z_{\emptyset}, \psi)$ is a partial function,

$$\iota: \quad Z_{\emptyset} \rightarrow Z_{\emptyset}$$

where the domain of ι is defined by its corresponding b-expression, $b_{\iota}(z)$, i.e. $\iota(z_1)$ is defined iff $b_{\iota}(z_1)$ is satisfied.

Definition 3.13

A *primitive deletion function*, or a d-function, δ, for a primitive data type $t=(Z_{\emptyset}, \psi)$ is a partial function,

$$\delta: \quad Z_{\emptyset} \rightarrow Z_{\emptyset}$$

where the domain of δ is defined by its corresponding b-expression, $b_{\delta}(z)$, i.e. $\delta(z_1)$ is defined iff $b_{\delta}(z_1)$ is satisfied.

Let us now define the characteristic set of operations, namely CDSO, for a primitive data type.

Definition 3.14

A CDSO set for a primitive data type $t=(Z_{\emptyset}, \psi)$ is $\psi=I_{U}D$, where

$$I=\{i_j\}_{j\leq J}$$

$D=\{d_\kappa\}_{\kappa\leq K}$

- for every i_j there exists a d_κ, and vice versa, such that for $z \varepsilon Z_\emptyset$:

  $d_\kappa(i_j(z))=z$ if $i_j(z) \varepsilon Z_\emptyset$,

AND

  $i_j(d_\kappa(z))=z$ if $d_\kappa(z) \varepsilon Z_\emptyset$.

We define the reachability relation for primitive data types in the same manner as we did for the non-primitive data types. As before the symbol "|-" will be employed to denote this relation. If $i_2\phi$ corresponds to the letter "B", then ALL$|-_{\{i_2,d_2\}}$BALL indicates BALL is reachable from ALL wrt either i_2 or d_2. Note that ALL is also reachable from BALL wrt d_2 i.e. BALL$|-_{\{i_2,d_2\}}$ALL.

3.4.3- Primitive Data Types as Posets

In this section we shall present some informal discussions about the primitive data types and their similarities with the non-primitive data types. In most cases, due to their close resemblance to the non-primitive data types, the results of the preceding sections may be applied directly to the primitive data types.

The word algebra for the case of the primitive data types is defined in the same manner as before. Once again we are only interested in the MF's for every term. For example, there are many ways of representing the string ALL.

For example, $d_2(i_2(i_1(i_{12}(i_{12}(\phi)))))$ or $i_1(i_{12}(i_{12}(\phi)))$ are among many ways of representing this string. Obviously every configuration of a data type may be represented by a series of insertion and/or deletion functions. As before we are only interested in a unique representation of each configuration. First let us define the minimal form of a configuration, z, of a primitive data type. This is defined recursively. Let $t=(Z_\emptyset,\psi)$ be a primitive data type with i-functions: $i_1, i_2, \ldots, i_n$, and d-functions: $d_1, d_2, \ldots, d_m$ in ψ. Then the *minimal forms*, or *irreducible* terms, of the word algebra of t are defined as follows:

either

1) ϕ is a MF,
2) $i_j(z)$ is a MF if z is a MF and $\bar{i}_j(z)$ is defined, $1 \le j \le n$

or

1) ϕ is a MF,
2) $d_j(z)$ is a MF if z is a MF and $\bar{d}_j(z)$ is defined, $1 \le j \le m$

Using the result of theorem 3.2, every term may be condensed into an irredundant deletion-free (or insertion-free) form using the reduction rule:

- if $t_1, t_2 \varepsilon W_t$ and $\bar{t}_1, \bar{t}_2 \varepsilon Z_\emptyset$ and $t_2=d_1(i_1(t_1))$ or $t_2=i_1(d_1(t_1))$, then $t_2 \Rightarrow t_1$.

The reachability relation implies two configurations of a data type are reachable from one another wrt a finite number of application of zero or more constructors. Since reachability as defined before is a reflexive relation then

every configuration is reachable also from itself wrt zero number of constructors. Similarly, as for the case of non-primitive data types, let >> denote the monotonic reachability relation(MRR). The >>-relation denotes reachability wrt a set of either monotonically increasing *or* monotonically decreasing functions, but not both. Looking back at the **string** example given earlier, BALL>>ALL but ALL>≯BALL, where >> denotes MRR wrt the i-functions only.

As in the case of the non-primitive data types, the >>-relation defines a partial ordering(po) on the set $\mathbf{W}_\theta$ of a primitive data type. Furthermore, it may be shown that the primitive data types are also lattice structures under the >>-relation.

For every primitive data type there are a number of insertion and/or deletion functions without which one cannot generate (or reach) all the desired configurations. It is precisely this set of functions that is of interest to us. For example for **string** data type, $i_1(\phi)$, or A, is one such term that is not reachable in the absence of i_1. The definition of a *generator function* for a primitive type follows from definition 3.10 given earlier.

The characteristic set of DSO's presented above defines the ground axioms for all the primitive data types. To specify a specific primitive data type our task reduces to defining the boundaries of the desired operations, by way of

the b-expressions, such that, nothing but the desired set of configurations would result.

Finally we extend the definition of i- and d-functions, for a primitive data type, for those cases where the primitive data type is at level zero of a data structure. Note that every i- and d-function associated with level zero would then become an element of ψ, where $t=(Z_L,\psi)$ and Z_L terminates with primitive nodes.

<u>Definition 3.15</u>

Let $t=(Z_L,\psi)$ be a data type and Z_L be terminated with primitive nodes. Let $i_\emptyset$($d_\emptyset$) denote an i-function(d-function) at level 0. For z, $z'\varepsilon Z_L$ and $i_\emptyset \varepsilon \psi$,[24]

$$i_\emptyset(z,0,k)=z'$$

iff z' is exactly the same as z except that

$$C(S'_{1qk})=i_\emptyset C(S_{1qk}) \text{ for some } q$$

Also $d_o(z,0,k)=z'$ iff z' is exactly the same as z except that

$$C(S'_{1qk})=d_\emptyset(S_{1qk}).$$

In chapters 2 and 3 we identified the essential elements that are required in order to isolate the desired behaviour of a data type. In the following chapter we shall

---

[24] Intuitively, the equation implies that change the content of the node <1qk>, say m=C(<1qk>), to the new configuration $i_\emptyset$(m). It is noteworthy that for this change to happen, the b-expression for $i_\emptyset$ (or $d_\emptyset$) must of course be satisfied with m.

examine the concepts of "completeness" and "soundness" of the specification of data types. We shall also look at ways of generating new data types, using the existing ones, in order to get a data type of a different behaviour.

CHAPTER IV

COMPLETENESS AND TYPE MANIPULATION OPERATIONS

In this chapter we shall examine the concepts of completeness and soundness of the specification of data types and show that every configuration that is a true configuration may be deduced and vice versa. The notion of "truth" of a term is introduced and used to examine the validity of the configurations that may be reached wrt the operations introduced in the enrichment process.

Another useful concept in specification techniques, for data types, is the capability of defining operations in such a way that two or more data types are combined to arrive at a new data type. This should be done in a manner that the newly-defined data type preserves the original properties of the constituent parts. Thus the characteristic operations of the newly-defined data type may be automatically defined in terms of the operation sets of the constituent data types. The "type manipulation operations" (TMO) are intended to facilitate the construction of new data types in terms of the existing ones. The latter may be user-defined or part of the system, i.e. system defined. In chapter 3 it was shown that data types may be modeled as

lattices. Each lattice structure is formed from a reachability relation equivalence class. The TMO's, therefore, may be considered to be operations on lattices.

4.1- Completeness and Soundness

There are many notions of "completeness" defined for different purposes in logic and other branches of mathematics. Logicians, for example, define a complete set of axioms as one with which every wff or its negation can be proved as a theorem, or one for which all models are isomorphic. Sometimes a complete set of axioms is related to the notion of "consistency." For instance, to say a set of wff's is complete is an equivalent phrase for a "maximally consistent" set of wff's. The latter notion indicates that by adding any wff, not an element of the set, to the given set, we would have an inconsistent set of wff's.

Our interest in completeness is that if a term is true in a data type t, then we must be able to reach this term. The converse of this is also important, i.e. if a term is reached, it must be the case that it is a true term. These notions, we shall refer to, as the completeness and soundness respectively.

To investigate the notions of completeness and soundness of a data type specification we need to define the concept of "truth" or "falsity" of a term. For example,

given a term $t=i(i(\phi))$, one may be interested to know under what circumstances, or what "values" of i and ϕ, is $\bar{t}$, the value of t, a legitimate configuration of a data type t. Thus for certain assignments of values to the function symbols, constants, variables etc., one can claim that $\bar{t}$ is an element of t. Or we may use the paraphrase: t *is true in* t whenever the assignment of the functions, constants and variables is understood; or using the terminology of chapter 3, $t \varepsilon$NET of t. To express the above concepts formally, we need to define the well formed formulas of our language.[25] Their interpretation will then be investigated.

Let g_K^n denote an n-ary function symbol. An individual variable or an individual constant is a *term*. If $\tau_1, \tau_2, \ldots, \tau_n$ are terms then $g_K^n(\tau_1, \tau_2, \ldots, \tau_n)$, $n \geq 1$, or ϕ is an *atomic term well formed formula (twff)*.

1) An atomic twff is a twff.
2) $g_K^n(\gamma_1, \gamma_2, \ldots, \gamma_n)$, $n \geq 1$, is a twff if γ_j, $1 \leq j \leq n$, are twff's.
3) Only those formulas obtained by a finite number of applications of 1) through 2) are twffs.

As an example of a set of twff's, one may consider the set of non-error terms, NET, of a data type. Recall that

---

[25] Note that these are not the wff's of the first order predicate logic described earlier. These are an extension of the terms of the word algebra that we talked about in chapter 3.

every $t \varepsilon NET$ is only composed of function symbols and constant symbols.

Now consider a set of twffs Γ. Let $\gamma \varepsilon \Gamma$. In the presence of a structure t and an assignment function s, a meaning may be attached to each twff γ; then we can say either "γ is true in t" or "γ is false in t." The former is denoted by $|=_t \gamma[s]$ where s is an assignment function (see below); the subscript t may be dropped when it is clear from the context. For example, if $\gamma \varepsilon NET$ then $|=_t \gamma[s]$. The assignment function s assigns values to the function and constant symbols in γ. In general, s is a function s: $Var \rightarrow Z_U N$, where Var is the set of all variables: namely l,k,i,.. to denote **N**, and z, z_1, z_2,... are individual variables to denote elements of **Z** where **Z** is a data structure.[26]

Let us now define what is meant by the satisfaction of a twff γ in the presence of a **"structure"** $t=(Z,\psi)$ and a function s. The symbol **t**, strictly speaking, is of course a data type. However it is also used to denote a **structure**, as described below, the context will remove any ambiguities. Let $t=(Z,\psi)$ and recall that $Z=\{z: z=(S,u)\}$, where every S is the union of S_l for every level l. For each level, S_l is itself composed of the sets S_{lk}. Thus when we talk about **t**, as a **structure**, we mean a family of structures, σ_{lk}, as

---

[26] z, z_1, z_2, ... may also be used as constant symbols. The context should remove any ambiguity.

defined in chapter 2. In addition each σ_{lk} is also equipped with the i-, d- and p-functions for that level, l. The universe of t is $Z_U S_U N^+$, where S is the set of all nodes, S_{lki}. As in chapter 2, implicit in our notation of the structure, is the presence of the traversal functions as well as any other function which may be necessary depending on the **structure** at hand.[27] Let $\bar{Z}\subseteq Z$ be the set of configurations that are reachable from ϕ wrt ψ. It is now possible to define "t *satisfies* γ *with* s", denoted by $|=_t\gamma[s]$ if γ is one of the following. Let ζ be a twff, χ be a term symbol, and let $z=\bar{s}(\zeta)$

1) $|=_t\chi[s]$ if $\bar{s}(\chi)\varepsilon\bar{Z}$.
2) $|=_t g^4(\zeta,l,k,i)[s]$ if $|=_t\zeta[s]$ and $g^4\varepsilon\psi_{s(l)}$ and *either* $S_{\bar{s}(lki)}\varepsilon CS^{z}_{\bar{s}(lk)}$ if g^4 is an i-function *or* $S_{\bar{s}(lki)}\varepsilon DS^{z}_{\bar{s}(lk)}$ if g^4 is a d-function. And if l<L, $\sigma_{l+1,k}$ satisfies $a_{l+1}(S_{l+1,q,k})$ with s.

Some explanations, regarding the above definition, are in order. Item (2) imposes the restriction that for a twff, $i(\zeta,l,k,i)$, to represent a true configuration of a data type t, it must be the case that the value of ζ itself is a true configuration of t and the value of the node represented by <l,k,i> is in the construction set of (S_{lk},u_{lk}). Where (S_{lk},u_{lk}) is a configuration at a level, l, in z.

---

[27] Every function symbol, $\underline{g}$, is assigned to a function g by the structure. Also $\bar{s}(lki)=(s(l),s(k),s(i))$, and $\bar{s}[g(\gamma_1,\gamma_2,\ldots,\gamma_n)]=g(\bar{s}[\gamma_1],\bar{s}[\gamma_2],\ldots,\bar{s}[\gamma_n])$.

If Γ is a set of twff's, then $\Gamma|=\gamma$ denotes that for every structure t and every function s such that t satisfies every element of Γ with s, t also satisfies γ with s. For a given t and s we may write $\Gamma|=_t\gamma[s]$ to denote that t satisfies every $\gamma_i\varepsilon\Gamma$ as well as γ with s.

In the following definition $\gamma[s]$ denotes $\bar{s}(\gamma)$, where γ is a twff. Also $\Gamma[s]$, where Γ is a set of twff's: $\{\gamma_i\}$, denotes the set $\{\gamma_i[s]\}$. Finally $\Gamma|-_\psi\gamma[s]$ denotes $\Gamma[s]|-_\psi\gamma[s]$; the latter means $\gamma[s]$ is reachable from one or more $\gamma'[s]\varepsilon\Gamma[s]$ wrt ψ where the underlying logical **structure** is $t=(Z,\psi)$.

Definition 4.1

A data type $t=(Z,\psi)$ *contains* a set of twff's, Γ, if there exists an assignment function s such that for every $\gamma\varepsilon\Gamma$, $\phi|-_\psi\gamma[s]$.

∎

Let $t=(Z,\psi)$ and $t'=(Z',\psi')$; we say that t contains t' if t contains W'_θ-{oft,uft}, up to renaming. Note that the above definition implies that for any data type $t=(Z,\psi)$, t always contains itself. It may also be concluded that for two data types $t=(Z,\psi)$ and $t'=(Z',\psi')$, if $\bar{Z}'\subseteq\bar{Z}$ then t contains t'. Or, in other words t is a correct extension of t'

4.1.1- Deduction of terms

The notion of reachability introduced earlier in

chapter 3 deals with the values of the twff's. In other words if γ and γ' are twff's then in the presence of a data type we may say that $\gamma[s]$ is reachable from $\gamma'[s]$. On the other hand, if we are not concerned with the underlying structure but interested only in knowing if a term is "deducible", i.e. reachable in a syntactic sense, from another term, we must introduce some rules of inference. Let Γ be a set of twff's, γ, and let β be a possibly empty, set of ground instances (sentences) of b-expressions, $b^{l}_{\iota_\gamma}(m)$, $b^{l}_{\delta_\gamma}(n)$, and a-expressions $a^{l}_{\gamma}(r)$ associated with $\gamma \varepsilon \Gamma$, where m, n, and r are some node constant symbols. Let us denote the pair (Γ,β) by Γ_β. Let i and d denote i- and d-function symbols and l, k, and m be integer constant symbols. The rules of inference are as follows: let $\gamma \varepsilon \Gamma$ contain no variable symbols,

1) if $\gamma \varepsilon \Gamma$ and $b^{l}_{\iota_\gamma}(\langle l,k,m \rangle) \varepsilon \beta$ and $a^{l+1}_{\gamma}(\langle l+1,q,k \rangle) \varepsilon \beta$ then
$$\gamma \to i(\gamma,l,k,m)$$
2) if $\gamma \varepsilon \Gamma$ and $b^{l}_{\delta_\gamma}(\langle l,k,m \rangle) \varepsilon \beta$ then
$$\gamma \to d(\gamma,l,k,m)$$
3) if $\alpha = i(\gamma,l,k,m) \varepsilon \Gamma$ and $b^{l}_{\delta_\alpha}(\langle l,k,m \rangle) \varepsilon \beta$ then
$$\alpha \to \gamma$$
4) if $\alpha = d(\gamma,l,k,m) \varepsilon \Gamma$ and $b^{l}_{\iota_\alpha}(\langle l,k,m \rangle) \varepsilon \beta$ then
$$\alpha \to \gamma$$

We say that γ is *deducible* from Γ_β, denoted $\Gamma_\beta \mid\!-\gamma$,[28] if $\gamma \varepsilon \Gamma$, or γ can be obtained by a finite number of applications

---

[28] Note that the subscript ψ of $\mid\!-$ is missing to denote deduction rather than reachability.

of the rules of inference. In other words, if there exists a finite sequence of zero or more inferences 1-4:

$$\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$$

where the left hand side (with respect to $\rightarrow$) of each α_i, $1 \le i \le n$, is either a term from Γ, or the right hand side of an α_j, $1 \le j \le i$, such that the right hand side of α_n is γ.

Theorem 4.1

Let $t=(Z,\psi)$ be a data type; let γ and Γ_β be a twff and a set of twff's, with the associated set of b-expression and a-expression sentences β, respectively. If b^1_ι, b^1_δ, and a^1 denote the b- and a-expressions, at level 1, associated with t, then let β be the set of sentences $b^1{}_{\iota_\alpha}(m_1)$, $b^1{}_{\delta_\alpha}(n_1)$, and $a^1_\alpha(r_1)$, where m_1 and n_1 are the constant symbols corresponding to the elements of the CS's and DS's of $\alpha[s]$, at level 1, respectively, and r_1 are the constant symbols in $R^{\alpha[s]}_{1k}$ for all k, where $\alpha \varepsilon \Gamma$. Then $\Gamma |-_\psi \gamma[s]$ iff $\Gamma_\beta |- \gamma$ for some s.

Proof

If $\Gamma -_\psi \gamma[s]$ then $\gamma[s]$ is reachable from one or more $\alpha[s] \varepsilon \Gamma[s]$. Therefore γ is some sequence of i- and d-functions "concatenated" with ϕ. But if γ is such a sequence then $\gamma |- \phi$ and $\phi |- \gamma$. Now if $\alpha[s] \varepsilon \Gamma[s]$ then $\alpha |- \phi$, therefore $\alpha |- \gamma$. Now let $\Gamma |- \gamma$, then there exists a term $\alpha \varepsilon \Gamma$ such that $\alpha |- \gamma$ since at every step of deduction the b-expression for insertion (deletion) must be satisfied plus the fact that the starting configuration, in this case α,

must be satisfied, then $\alpha[s]$ itself is reachable from ϕ and γ is, therefore, reachable from $\alpha[s]$, i.e. $\alpha[s]|-_{\psi}\gamma[s]$. The latter implies that $\Gamma|-_{\psi}\gamma[s]$ since $\alpha\varepsilon\Gamma$.

Theorem 4.2-*completeness theorem*

Let t be an arbitrary data type with the b-expressions b^1_ι and b^1_δ, and a-expressions a^1 at each level 1. Let Γ_β be as defined in theorem 4.1, then $\Gamma|=\gamma$ implies $\Gamma_\beta|-\gamma$.

Proof:

Let us select an arbitrary data type $t=(Z,\psi)$ and show by an inductive argument that if $\Gamma|=_t\gamma[s]$ then $\gamma[s]$ is reachable from ϕ wrt ψ, since, by theorem 4.1, if $\gamma[s]$ is reachable wrt ψ then γ is deducible from ϕ.

Since t satisfies γ with s we may assume that $\gamma[s]=i^k_q\phi$ where i^k_q is an i-monotonic sequence and $I=\{i_1,i_2,...,i_n\}\subseteq\psi$. If $\Gamma|=_t\gamma[s]$, then $\Gamma[s]|=_t i^k_q\phi$ for some sequence i^k_q. The latter may be written as $\Gamma[s]|=i_j i^{k-1}_{q'}\phi$, for some $1\leq j\leq n$. Since t satisfies $\gamma[s]$, then t satisfies $i_j i^{k-1}_{q'}\phi$. By the definition of satisfaction, therefore $i^{k-1}_{q'}\phi$ is also satisfied, and so is $i^{k-2}_{q''}\phi$ and so on. Note that in the above argument we have assumed that $\gamma[s]$ is in MF composed of an i-monotonic sequence. The presence of the d-functions as well as i-functions does not alter the above argument. All that is needed is to replace the symbol i with d to denote their presence.

Thus $\gamma[s]=i_j i_{j_1} i_{j_2} \ldots i_{j_J}\phi$, for j, $j_1,..,j_J \le n$, therefore, by the definition of reachability, the latter equation denotes:

$$\phi |-_{\psi} \gamma[s] \;\ldots.. \;(1)$$

Let us now consider a term $\gamma' \varepsilon \Gamma$ and let d^m be, elements of ψ, forming an inverse sequence of i^m, such that if $\gamma'[s]=i^m\phi$ then $\phi=d^m\gamma'[s]$.[29] Therefore

$$\gamma'[s] |-_{\psi} \phi \;\ldots. \;(2)$$

Since $|-_{\psi}$ is transitive, therefore (1) and (2) yield $\gamma'[s]|-_{\psi}\gamma[s]$. But γ' was chosen arbitrarily, hence $\Gamma|-_{\psi}\gamma[s]$. But, by theorem 4.1, the latter implies that $\Gamma_{\beta}|-\gamma$.

The above theorem ensures that all the twff's that are true in some $t=(Z,\psi)$ are deducible and hence their corresponding configurations are also reachable from ϕ wrt ψ.

The converse of the above (completeness) theorem is the *soundness theorem* presented below.

Theorem 4.3-*soundness theorem*

Let $t=(Z,\psi)$ be an arbitrary data type; let Γ_{β} be a set of twff's and β as defined in theorem 4.1, then if $\Gamma_{\beta}|-\gamma$ then $\Gamma|=\gamma$.

---

[29] Note that since t satisfies γ with s, once again we may write $\gamma'[s]=i^m\phi$ by definition of satisfaction.

<u>Proof</u>

Once again we take an arbitrary data type t and show that if $\Gamma|-_{\psi}\gamma[s]$ then $\Gamma|=_{t}\gamma[s]$; the result would then follow from theorem 4.1.

Let z denote $\gamma[s]$. If z is reachable from every element, $\gamma[s]$, of $\Gamma[s]$ wrt ψ, then if $\gamma'[s]$ is satisfied with t then z is reachable from ϕ wrt ψ since $\gamma'[s]$ must be reachable from ϕ. Let us also assume that the configuration z is reachable after n applications of the operations f_1, $f_2,\ldots,f_n \varepsilon \psi$, not necessarily distinct. By an inductive argument on n we show that: z is true in t(i.e. t satisfies γ with s), where $z=f_n(\ldots(f_2(f_1(\phi))\ldots)$. But ϕ is true in t, by definition. Also $f_1(\phi)$ is true in t, otherwise either a p- or b- expressions would be violated. But if this was the case, and f_1 is one of the many possible operations needed to reach z, then there would be no operations to make z a reachable configuration. This is contrary to our assumption. Assume $f_{n-1}(\ldots(f_1(\phi),\ldots)$ is true in t; by the same arguments as above there must be a operation f_n such that $z=f_n(f_{n-1}(\ldots(f_1(\phi),\ldots)$ and therefore z is true in t. But then if this is the case, then $Z|=f_n(f_{n-1}(\ldots(f_1(\phi),\ldots)$ is true by definition. Hence z is true in t, therefore $Z|=z$. But since $\Gamma[s]\subseteq Z$, therefore $\Gamma[s]|=_{t}z$, i.e. $\Gamma|=_{t}\gamma[s]$.

Now since every configuration corresponding to a true twff is reachable from ϕ, we can state the following.

Theorem 4.4

Let $t=(Z,\psi)$ be a data type; if γ is a twff which is true in t with some s, then $\gamma[s]$ is reachable from ϕ wrt ψ.

Theorems 4.3 and 4.4 imply that if a configuration, $\gamma[s]$, is reachable from one or more configurations of a data type t, then it must be the case that t satisfies γ with s and vice versa.

Assume a family of p-expressions, b-expressions, and a-expressions, then the sets I and D can be uniquely determined, since I and D are uniquely characterized by p-, b-, and a-expressions (see the definitions 3.2 and 3.3 of i- and d-functions). The p-functions are also characterized by their corresponding a-expressions. By theorem 4.4, (ϕ,ψ) is sufficient to enumerate all the elements of a data type (Z,ψ). Hence the base predicate set is sufficient to define $t=(Z,\psi)$. Thus to specify the "behaviour" of a data type one may only specify the p-expressions, b-expressions, and the a-expressions at every level.

4.2- Type Manipulation Operations

Burstall and Goguen [BUR 77] argue that : "as soon as the theories get to be interesting they become incomprehensible. We wind up with a large set of equations

that no one can understand and which are almost certainly wrong. So we must build our theories up from *small* intelligent pieces. For this we need

i) The ability to write (small) explicit theories,

ii) Four operations on theories:

1) Combine
2) Enrich
3) Induce
4) Derive

enabling us to build up theory expressions denoting complex theories."

Our principal objective, in the use of type manipulation operations, is to provide a framework for the above operations and more. The parameterization of data types will also follow naturally from this approach.

<u>Definition 4.2</u>

The *embed* of a Σ-structure $\Sigma'_{L'} \varepsilon Z'$, where Z' is a data structure terminated with SSN, and a data structure $Z''_{L''}$, terminated with SSN ($\iota=1$), or with primitive nodes ($\iota=\emptyset$), is a set of Σ-structures Z, denoted $Z_L=\Sigma'_{L'} \times Z''_{L''}$; where $L=L'+L''$ and

$$Z_L=\{\Sigma_L=(S,u) \ : \ S=\bigsqcup_{\iota<l\leq L} S_l \text{ and } u=\bigsqcup_{1<l\leq L} u_l \quad \text{such that}$$

$$\Sigma=(\sigma,\upsilon)=(\bigsqcup_{L''<l\leq L} S_l \ , \bigsqcup_{L''<l\leq L} u_l) \simeq \Sigma'_{L'} \text{ AND}$$

$$\forall S_{L''+1,k,i} \varepsilon \sigma \ (C(S_{L''+1,k,i}) \simeq \Sigma'' \text{ where } \Sigma'' \varepsilon Z'') \ \}$$

We may represent each element $z \varepsilon Z$ by $z=(\Sigma', \{z^{\cdot\cdot}_m\}_{|S'_1|})$ where the latter indexed set contains elements $z^{\cdot\cdot}_m \varepsilon Z''$ such that each $z^{\cdot\cdot}_m$ is the configuration contained in a node of Σ corresponding to $S'_{1km} \varepsilon S'_1$, in other words $m \varepsilon \{1,2,\ldots,|S'_1|\}$.

The *embed* of two data structures Z' and Z'' is denoted by $Z=Z' \times Z''$ where Z is the set of Σ-structures:

$$Z = \bigsqcup_{\Sigma' \varepsilon Z'} \Sigma' \times Z''$$

Reminder: a subscripted operation symbol o_l denotes an operation defined for the l th level structure of a data structure.

Definition 4.3

The embed of two data types $t'=(Z'_{L'}, \quad , \psi')$ and $t''=(Z''_{L''}, \psi'')$ is denoted by $t=(Z_L, \psi)=t' \times t''$, such that

$Z_L = Z'_{L'} \times Z''_{L''}$ and $\psi=\{I, D, P\}$ where

$I=\{i_l: L''+1 \leq l \leq L''+L' \ \& \ i_{l-L''} \varepsilon I' \text{ or } \emptyset \leq l \leq L'' \ \& \ i_l \varepsilon I'' \}$

$D=\{d_l: L''+1 \leq l \leq L''+L' \ \& \ d_{l-L''} \varepsilon D' \text{ or } \emptyset \leq l \leq L'' \ \& \ d_l \varepsilon D'' \}$

$P=\{p_l: L''+1 \leq l \leq L''+L' \ \& \ p_{l-L''} \varepsilon P' \text{ or } \emptyset \leq l \leq L'' \ \& \ p_l \varepsilon P'' \}$

Consider now the word algebra of the embed of two data types. The embed of t and t' is composed of the configurations which may be represented in the form of $(t, (t_1, t_2, \ldots, t_n))$ where n is the number of nodes at level 1 of the configuration represented by t. Consider two terms of the word algebra of MF's of $t \times t'$: $\mathit{t}=(t, (t_1,t_2,\ldots,t_n))$ and $\mathit{t}'=(t', (t'_1,t'_2,\ldots,t'_m))$, where $n \geq m$; $\mathit{t} >> \mathit{t}'$ iff $t >> t'$

and $(t_1, t_2, \ldots, t_n) >> (t'_1, t'_2, \ldots, t'_m))$ where the latter means $t_1 >> t'_1$, $t_2 >> t'_2, \ldots, t_{n-i} >> t'_m$ for some $i \varepsilon \mathbf{N}$. For the sake of notational convenience we shall consider the second term in each pair: $t=(t, (t_1, t_2, \ldots, t_n))$ collectively, so that t may be denoted as $(t, \tau^{1 \to n})$. Thus

$$(t_1, \tau^{1 \to n}) >> (t_2, \tau^{1 \to m}) \text{ iff } t_1 >> t_2 \text{ and } \tau^{1 \to n} >> \tau^{1 \to m}$$

Note that the above definition of >> is consistent with the definition of >>-relation on W_θ given earlier in chapter 3. The MF's for the configurations represented by t and t' are unique so that if t and t' are translated into their corresponding MF's, t_{MF} and t'_{MF} respectively, then $t_{MF} >> t'_{MF}$ iff $t >> t'$. The latter statement is true since, intuitively, for $t_{MF} >> t'_{MF}$, it must be the case that the corresponding "components", viz. structures contained in each node, must be monotonically reachable.

<u>Theorem 4.5</u>

Let ω and ω' be congruence relations on (W_t, ψ) and $(W_{t'}, \psi')$, where $t=(Z, \psi)$ and $t'=(Z', \psi')$. Let t_1, $t_2 \varepsilon W_t$ and $t_1^{1 \to m}$, $t_2^{1 \to n}$ be elements of $W_{t'}$, then the definition of the relation ω_x such that

$$(t_1, t_1^{1 \to m}) \omega_x (t_2, t_2^{1 \to n}) \text{ iff } t_1 \ \omega \ t_2 \ \& \ t_1^{1 \to m} \omega' t_2^{1 \to n}$$

where $t_1^{1 \to m} = \{t_{11}, t_{12}, \ldots, t_{1m}\}$ and $t_2^{1 \to n} = \{t_{21}, t_{22}, \ldots, t_{2n}\}$, gives a congruence relation on $(W_{t \times t'}, \psi_U \psi')$.

<u>Proof</u>

First we show that the following three properties hold for ω_x:

symmetry:

Show that:

$(t_1,t_1^{1\to m})\omega_x(t_2,t_2^{1\to n})\to(t_2,t_2^{1\to n})\omega_x(t_1,t_1^{1\to m})$

LHS is true iff $t_1\ \omega\ t_2$ and $t_1^{1\to m}\ \omega'\ t_2^{1\to n}$ iff $t_2\ \omega\ t_1$ and $t_2^{1\to n}\ \omega'\ t_1^{1\to m}$ iff RHS.

reflexivity:

Show that $(t_1,t_1^{1\to m})\ \omega_x\ (t_1,t_1^{1\to m})$. This is true iff $t_1\ \omega\ t_1$ and $t_1^{1\to m}\ \omega'\ t_1^{1\to m}$. The latter is clearly true, hence ω_x is reflexive.

transitivity:

Show if $(t_1,t_1^{1\to m})\omega_x(t_2,t_2^{1\to n})$ and $(t_2,t_2^{1\to n})\omega_x(t_3,t_3^{1\to k})$ then $(t_1,t_1^{1\to m})\omega_x(t_3,t_3^{1\to k})$

$(t_1\ \omega\ t_3)$ iff $(t_1\ \omega\ t_2)$ and $(t_2\ \omega\ t_3)$

$(t_1^{1\to m}\ \omega'\ t_3^{1\to k})$ iff $(t_1^{1\to m}\ \omega'\ t_2^{1\to n})$ and $(t_2^{1\to n}\ \omega'\ t_3^{1\to k})$ iff

$$(t_1,t_1^{1\to m})\ \omega_x\ (t_3,t_3^{1\to k})$$

Finally we have to show that the substitution property holds, note that we only deal with "unary" functions.

$(t_1\ \omega\ t_2)\to f\ (t_1)\ \omega\ f(t_2)$, and

$(t_1^{1\to m}\omega'\ t_2^{1\to n})\to f'(t_1^{1\to m})\ \omega'\ f'(t_2^{1\to n})$ then

$(t_1,t_1^{1\to m})\omega_x(t_2,t_2^{1\to n})\to$

$f_x((t_1,t_1^{1\to m}))\ \omega_x\ f_x((t_2,t_2^{1\to n}))$

iff $(t_1,t_1^{1\to m})\ \omega_x\ (t_2,t_2^{1\to n})$

But the latter conditions are true by definition, hence the substitution property holds and ω_x is a congruence relation on $(W_{t\times t'}\ ,\ \psi_U\psi')$.

It can be concluded that the υ-relation is preserved by the $\times$-operator.

In some cases the structure at the lower levels of a Σ-structure presents us with extraneous detail to the extent that we may only be interested in the structures of the top few levels. Thus it is beneficial to define a mechanism by means of which, structures at lower levels can be removed. This facility is particularly helpful in cases where the Σ-structures are terminated with primitive nodes. A reduction operator, /, is employed for this purpose. The following definition identifies a Σ-structure which is formed by removing the level zero, or one, of a Σ-structure.

Definition 4.4

The *red-structure* of an L-level Σ-structure Σ_L, $L-\iota \geq 1$, terminated with level ι, $\iota=0$ or 1, is an $L-\iota$ level structure $\Sigma'_{L-\iota}$ terminated with SSN such that each node of $\Sigma'_{L-\iota}$, $S_{l'k'i'}$ is defined as follows:

Let S_{lki} denote the nodes of Σ_L, then for all values of l, k, and i, if Σ_L terminates with SSN (i.e. $\iota=1$) then

$<l',k',i'>=<(l-1),k,i>$ AND $l' \geq 1$ ... (1)

else $<l', k', i'>=<l, k, i>$

N.B. For the resulting red-structure, the nodes with $l'=1$ support $\Sigma_{undefined}$. Also the /-operator is undefined for

Σ_L, when $L \le 1$, terminated with SSN, since l' becomes less than 1 in equation (1) above.

Example

Consider the reduction operator /, applied to the 3-level structure G twice; the result is depicted in Figure 4.3.

Thus the reduction operator, /, is a morphism such that if /(t)=t' and if t is the product $t_1 \times t_2 \times \ldots \times t_J$, then $t' = t_1 \times t_2 \times \ldots \times t_{J-1}$ where t_J is either a 1-level data type at level one and the structure terminates with SSN, or t_J is a primitive data type.

In the above example of figure 4.1, the result of applying / to G yields the red-structure of G. The reduction of a Σ-structure may be extended to define the reduction of a set of Σ-structures, or a data structure. The *reduction* of a data structure Z_L is a data structure Z'_L such that for every $z \in Z$, there exists a $z' \in Z'_{L'}$ such that $z' = /(z)$, and there are no other elements in $Z'_{L'}$. If $L \le 1$ AND Z_L is terminated with SSN, OR $L < 1$, then $Z'_{L'}$ is undefined.

Definition 4.5

Let $t = (Z, \psi)$ be a data type such that either $L > 1$, or if $L = 1$ then Z_L terminates with primitive nodes. A *reduction(red) operator* /, is a partial mapping such that $/(t) = t' = (Z', \psi')$, and Z' is the reduction of Z, and

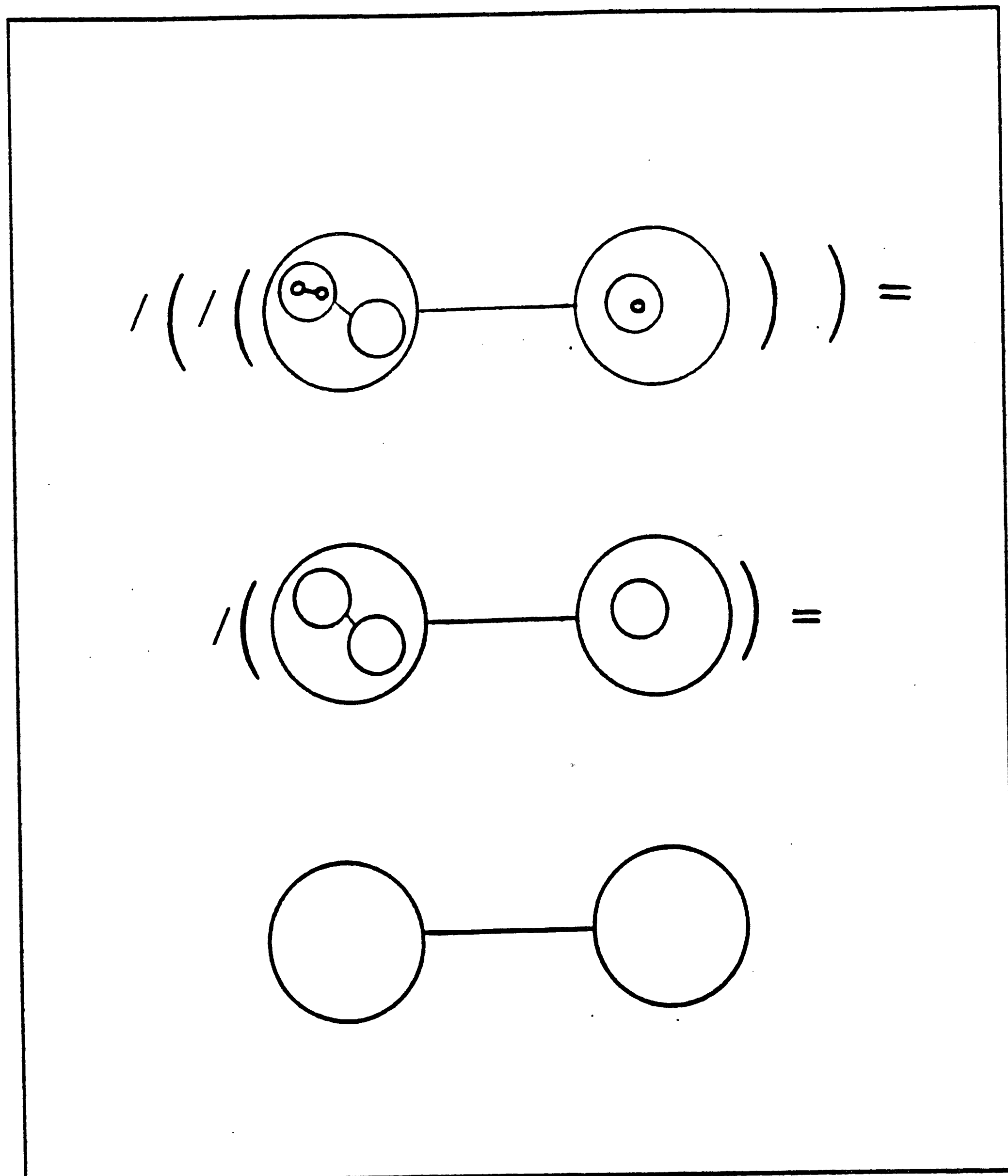


Figure 4.1
Removal of Inner Structures by /-function.

$$\psi' = I-\{i_l: \begin{array}{l} l=1 \text{ if } Z \text{ terminates with SSN,} \\ l=\emptyset \text{ otherwise} \end{array}\} \cup$$

$$D-\{d_l: \begin{array}{l} l=1 \text{ if } Z \text{ terminates with SSN,} \\ l=\emptyset \text{ otherwise} \end{array}\} \cup$$

$$P-\{p_l: \begin{array}{l} l=1 \text{ if } Z \text{ terminates with SSN,} \\ l=\emptyset \text{ otherwise} \end{array}\}.$$

The red-operator may be extended to remove more than one level of a data type rather than just the lowest level. Finally it is noteworthy that the reduction operator performs the reverse operation of the product operation in the sense that /(t×t')=t where t' is a data type with a single-level data structure.

In order to make our model more susceptible to automatic programming, the TMO's are designed such that the resulting data type preserves the attributes of the operand data types. As a result it would be possible to automatically determine the characteristic functions of the resulting data types without much difficulty. Another advantage of the TMO's is that it allows complex data types to be safely and correctly built using the more common and better understood data types. Hence break-down of complex structures, a valuable tool in structured programming, is strongly encouraged.

The type manipulation operations are of valuable help to the user of an abstract data type language. Our experience is that, because of the presence of the graph structures, one requires less effort to specify a data type using our approach over any one of the algebraic or axiomatic techniques.[30] The virtue of the type manipulation

---

[30] This is particularly true for simple data types, such as **stack**, **list**, and **tree**. As the ADJ group[GOG 78] has already pointed out, even for simple data types the algebraic technique can be trivially mis-specified or the specification may be incomplete and/or inconsistent.

operations, however lies in the fact that they enable us to define complex data types without much effort. This is because the resulting data type retains the original properties of the constituent, or the operand, data types. The TMO's remove the burden of defining generator operations, as well as specification of error instances, for the resulting data type, from the programmer to the machine. With a few simple data types at hand, one may employ TMO's in order to construct, or gradually build, more complex data types. Users may define auxiliary operations at every stage either "for convenience" or "change of behaviour." This will be pursued in the next chapter where we shall discuss one more TMO, namely the "enrichment" operation as well as the notions of equivalence and error.

CHAPTER V

ENRICHMENT, EQUIVALENCE AND ERROR

We shall investigate the enrichment process in the next section and examine the validity of the enrichments in general. Some conditions will be developed in order to signal "illegal enrichments." The concept of equivalence of data types will also be discussed. This subject is of particular interest where an implementation of a data type is sought. Finally the "error" criteria will be examined.

5.1- Enrichment of Data Types

Enrichment of the data types is the process by which new operations are introduced. Enrichment may therefore result in a change of behaviour of a data type. Consequently it may be employed in order to realize one data type in terms of another. There are also cases where enrichment does not alter the characteristic of the original data type. Under these circumstances enrichment is merely used for convenience. Enrichment is particularly useful in implementation process of data types [NOU 79]. That is the new operations are used to introduce new configurations resulting in a change of behaviour of the data type. At

this stage we would like to point out some of the deficiencies of the past techniques regarding the enrichment of the data types.

In the process of enrichment it is very easy to distort the behaviour of a data type. Consider, for example, the **stack** data type. One can easily enrich the **stack** by a "non-constructor" operation: *bottomnode*, say, to peek at the bottom node of the **stack** configurations. This is a distortion of the **stack** behaviour and yet it may be legal to define this type of operations in cases where there are no axioms to prevent the user. If the introduction of the above operation, by the user, is intentional then indeed there is no problem. However if such operation is carried out inadvertently, which is more likely in a more complicated data type, the consequences are obviously undesirable.

5.1.1- Changing Behaviour by Enrichment

A problem of different nature also related to the above example is that of the correctness proof methodology of Goguen et. al. [GOG 78]. Their approach is based on cannonical term algebras (cta) and the concept of constructor signatures. Briefly, two data types are assumed to be the same up to renaming if their cta's are isomorphic. The idea advocated is that: "to know a data type is to know its constructors." By the above stack example it is immediate that this is not always true. Consider the

original **stack** (a normal **stack** data type) and the **modified-stack**, with the new operation *bottomnode*. Note that *bottomnode* is not a constructor operation, both of these "**stacks**" have the same cta and constructor signatures. However they do not exhibit the same "behaviour."

The introduction of the probe function, in the characteristic set of DSO, obviates this problem. In the next section it will be shown that the **modified-stack** is no longer equivalent to the original **stack**, since their a-expressions are not the same. Hence any attempt made, by a user, to add an illegal operation of this nature would be immediately detected and prevented. Any other operation that does not distort the original behaviour may be added.

The question is then- if we are not allowed to change the characteristic set how can we realize one data type in terms of other data types? The answer is that certain mechanisms must be built-in to signal the change of characteristics in the process of enrichment. In an **array**, for example, introduction of *bottomnode* does not cause a change of behaviour of the **array** since we already have access to the bottom node (first node inserted) anyway. Thus, the question is how can we detect this change or no-change in behaviour? By enriching the operations of the characteristic set we either relax certain conditions or restrict them. To detect when a transition in behaviour

occurs, we adopt the following approach based on the three base predicate expressions characterizing a data type.

The change of behaviour, if any, which may result from an enrichment may be detected by examining if the newly introduced functions are in line with the p-, b-, and a-expressions when they are evaluated. This is explained below in more detail. Let us consider a **stack** data type. The construction sets for it is determined by its b-expression. So that every node to be inserted must satisfy the related b-expression $b_l(i)=\forall j(j\leq i)$. Thus the domain of the i-function is limited to that of the top node for every given configuration. Clearly to introduce another insertion function i' to allow insertion at the "bottom-end" of the **stack** would cause a change of behaviour. As a result the b-expression would not be satisfied with such an insertion. Similar arguments are also applicable in the case of the deletion function.

In general, there are two categories of enrichment. Firstly a data type may be enriched merely for the sake of "convenience" rather than change of behaviour. A second category of enrichment is "enrich to change behaviour." That is to obtain a data type with a different characteristic to that of the original one. A more "priviledged" operation is now needed to change the behaviour of a data type.

It should be noted that enrichment is a type manipulation operation. Consider the "ENRICH" operator which may be performed on a data type t. In case of the enrichment for convenience the ENRICH operator evaluates to the original data type. If only the b-expression is changed, then the resulting data type ENRICH(t) would either contain t, i.e. t $\subseteq$ ENRICH(t),[31] or vice versa depending on whether the b-expression was made more "strict" or more "relaxed" respectively. For the cases of enrichment where the a-expression is changed, t and ENRICH(t) would be isomorphic but not "behaviourally isomorphic." In order to show that one data type mimics the behaviour of another one needs to demonstrate their "behavioural isomorphism."

Auxiliary operations(AO) are introduced in the next subsection. These are the class of operations that when added to a data type t the characteristic of t would not be changed; that is $ENRICH_{AO}(t)=t$ where the equality implies behavioural isomorphism (see later), and the subscript AO denotes the type of functions that t is enriched with.

5.1.2- Auxiliary Operations

So far we have only discussed three types of data structure operations, namely i-, d- and p-functions. Such functions are of primary importance since their role is to completely specify the behaviour of the desired data type.

---

[31] $\subseteq$ denotes sub-isomorphism.

There exists another class of operations for the purpose of users' convenience rather than essential to the specification. Such operations are referred to as The auxiliary operations (AO).

The auxiliary operations define the "class of programmed operations." They may be thought of as procedures in a programming language. For example we may define the attach_leftmost operation on the **binary tree** data type in order to insert a node to the leftmost node in a **binary tree** configuration. This operation does not change the characteristic of the **binary tree** it merely provides us with an extra tool in order to implement other, more complex, operations. In a sense, the auxiliary operations are "non-essential" to the extent that their absence or presence does not influence the underlying behaviour of the data type. AO's are also, in general, non primitive because they may be broken up into more primitive operations. For example consider an operation that performs parallel insertion into an element of the **binary tree** data type. That is it inserts one or more nodes into a configuration of the **binary tree** concurrently. Such an operation may sound to be a constructor operation. However, it may be implemented in terms of the more primitive serial i-functions of the **binary tree** hence it is not a generator function for the **binary tree**. Thus parallel insertion of the **binary tree** would be categorized as an AO. It will be shown later that the latter operation would not affect the

overall behaviour of the **binary tree** since the configurations of the **binary tree** would remain unchanged. On the other hand, introduction of a parallel operation of insertion in a **stack**, such as inserting more than one node at a time, would violate its associated b-expression and/or a-expression, resulting in an illegal enrichment.

In order to define the auxiliary operations, one would need a media in which such operations may be specified with clarity and without ambiguity. Such a facility is provided by the language of "equational theory." This is discussed next.

5.1.2.1- Equational Theory

Many basic properties of algebras are expressed by means of equations, i.e. ordered pairs (f_1,f_2) often represented as $f_1=f_2$ using the language of *equational logic* [TAR 66]. We shall be concerned with formulas of the type u=v where u and v are the extension of the twff's defined in chapter 4. These extended twff's, denoted as well formed sentences, will be described shortly.

The data structure operations are, in general, divided into the characteristic set ψ and the auxiliary set, **A**. We have already discussed the former type of operations extensively in chapter 3. The set **A** will be the subject of what follows.

The central idea is to define a set of operations in terms of the elements of the characteristic set of operations. The correctness of the auxiliary operations is guaranteed by the fact that the constituent parts of an AO are themselves valid operations of the data type. Therefore we need not worry about the "error" instances. These are already taken care of by the operations of ψ. In any case such error instances resulting from an auxiliary operation would not lead to a change of behaviour, it may only cause undesired restriction on the actual auxiliary operation being defined. For example we may define a parallel insertion operation on the **binary tree** using its more primitive i-function. The error instances of such a parallel operation is restricted to those of the primitive i-function, since the semantics of the parallel operation may be defined in terms of the sequential i-function. The parallel operation would neither add a configuration to the set of **tree** configurations nor will it take one away from it.

In order to determine if an auxiliary operation is a "legal" operation, in the sense that it does not alter the behaviour of the data type **t**, we define the "satisfaction" of the newly defined operations, viz. the AO's. First let us define what is meant by an auxiliary operation. An *auxiliary operation* (AO), defined for a data type **t=(Z,O)**, is a partial function f such that

$$f: A_1 \times A_2 \times \ldots \times A_n \rightarrow Z_U S$$

where A_ι, $1 \le \iota \le n$, is either **Z** or *S*, where *S* is the set of all the nodes S_{lkm} for $l,k,m \in \mathbf{N}$. An AO, f, is an *auxiliary constructor operation*(ACO) if the range of values of f is in **Z**, it is an *auxiliary probe function* if the range of the values of f is in *S*.

In order to enhance the power of specification for the auxiliary operations a new operator symbol is introduced, namely *ifthenelse*, its value, wrt a data type t, will be defined later in this section. With the introduction of this operator we need to extend the definition of our twff's. The extended version is referred to as the *term well formed sentences* (twfs). These are defined below. Let p^n_κ denote an n-ary predicate constant, for $\kappa \ge 1$ and $n \ge 0$. An *atomic term well formed sentence* is either a twff or constant symbols: z_1, z_2,... denoting the elements of **Z**, or constant symbols: c_1, c_2,... denoting elements of *S*.

1) An atomic twfs is a twfs.
2) *if* $p^m_q(\tau_1,\tau_2,...,\tau_m)$ *then* τ *else* τ'
   is a twfs if τ_ι, τ and τ' are twfs's.[32]
3) Only those formulas obtained by a finite number of applications of 1) through 2) are twfs's.

Note that the set of twff's is a subset of the set of all twfs's.

---

[32] We may also write "if π then z_1 else z_2" as an alternative for ifthenelse(π,z_1,z_2).

The satisfaction of twff's was defined earlier. It only defines the satisfaction of twff's only if they have a value in **Z**. However we would like to extend this definition so that if a twfs evaluates to a node value we also have a notion of satisfaction for it. Let us now define the satisfaction of those function symbols whose value is in *S*.

Let f be an auxiliary probe function symbol. We say t satisfies f with s, denoted by $|=_t f[s]$ iff the value of f, denoted by $\bar{f}$,[33] for any argument values regardless of the arity of f, is such that $\bar{f}=S_{lkm}$ and $a_l(S_{lkm})$ is satisfied with σ_{lk}, where a_l is the a-expression associated with level l of **Z**.

The definition of satisfaction of the twfs's containing the ifthenelse operator is defined next.

- $|=_t($*if* $p_1^n(\gamma_1,\ldots,\gamma_n)$ *then* γ *else* $\gamma')[s]$ if $|=_t\gamma[s]$ and $|=_t\gamma'[s]$.

We may define the value of the ifthenelse operator as follows. Let γ=if π then γ' else γ'' be a twfs, in the presence of a data type **t** and some assignment function **s**, the value of γ is **s**[γ'] if **t**, the corresponding **structure**, satisfies π with **s**, otherwise the value of γ is **s**[γ''].

We define an AO set, **A**, for a data type **t**=(**Z**,ψ) using a set of equations *Æ* as follows. Let eε*Æ* be $\xi=\zeta$, then e *defines* ξ subject to the following conditions:

---

[33] $\bar{f}(\tau_1,\tau_2,\ldots,\tau_n)$ denotes $\bar{s}[f(\tau_1,\tau_2,\ldots,\tau_n)]$.

1) ξ is a twff of the form $f^n(\tau_1,\tau_2,\ldots,\tau_n)$, where f^n is an n-ary function symbol, $n\geq 0$, and each τ_r is a term, i.e. either a variable or a constant symbol, and there are no other equations e'εÆ in which f^n occurs in the LHS of e'. And ζ is a twfs.

2) If $f^n(\tau_1,\tau_2,\ldots,\tau_n)$ is the LHS of eεÆ then f^n is distinct from the function symbols g, corresponding to elements of ψ, and ϕ.

The satisfaction of the newly defined operator ξ is defined as follows.

$$|=_t \xi[s] \text{ iff } |=_t \zeta[s].$$

For example, let **t** be the **tree** data type. We are interested in an operation, ins2, in order to insert two nodes in a **tree** configuration every time that ins2 is invoked. Let ins denote the primitive **tree** insertion, then we may define

ins2(z,<lki>,<lkj>)=ins(ins(z,<lki>),<lkj>). Thus

if $|=_t$ins(z,<lki>)[s] and $|=_t$ins(ins(z,<lki>),<lkj>)[s]

then $|=_t$ins2(z,<lki>,<lkj>)[s].

The above idea of defining new operations may be extended in order to define parallel, insertion and deletion, operations only if all the nodes to be inserted into, or deleted from, z are in the appropriate construction sets or the destruction sets of z. Recall that this is a decidable problem as shown in lemma 3.1. Consequently, such a powerful facility may readily be utilized without any

concern about the correctness of parallel operations. The ability to define such operations is most valuable in performance measures. For example, given the **binary tree** specification, it is possible to specify parallel insertion and deletion operations or both.

The following is an example of an APF.

Example

Consider a **binary tree** data type. We wish to define a probe function which always returns the nearest leftmost leaf node to a node m. Assume t∊**binary tree** and let $\mathbf{p}_1$ denote the left probe function of the **binary tree**.

leftmost(t,m)= *if* $\mathbf{p}_1$(t,m)=Ø *then* m
else leftmost(t, $\mathbf{p}_1$(t,m))

To get the leftmost leaf of the tree, use leftmost(t,1).

■

Recall that in chapter 3, we introduced the reduction of the terms of a data type; before leaving this section, it must be added that if ξ is defined by ζ, i.e. $\xi=\zeta$, then $\xi=>\zeta$, meaning ξ reduces to ζ. To this end, in the presence of the auxiliary operations, we have the extra axioms, the auxiliary equations, by means of which the terms may be reduced to their MF. Hence the MF for ins2 operation, defined above, is given by the MF of the RHS of the equation in terms of the primitive i-function of the **binary tree**.

Finally, we can define what we mean by a specification of a data type.

Definition 5.1

A *specification* of a data type is a pair $(\Pi, Æ)$ where Π is an indexed family of base predicate sets π_l, $\Pi=\{\pi_l\}_{l\varepsilon L}$, such that each π_l is the set of p-, b- and a-expressions, the base predicate set for level l, and L is the index set; and Æ is a set of auxiliary equations.

5.2- Equivalence of Data Types

In the algebraic approach of [GOG 78, NOU 79], the philosophy is that "to know a data type is to know its constructors." Clearly in the simple example of **stack** given in section 5.1.1 we showed a contradiction to this philosophy. Because of this, and the vital importance of data security, we have chosen to include the probe function in the characteristic set of operations of a data type. In here, we shall see that the two **stacks** of section 5.1.1 are not "behaviourally isomorphic" although they have the same configurations in their respective data structures.

The notion of "weak equivalence" is defined next; this will be followed by a stronger notion of equivalence, viz. the behavioural isomorphism or simply "equivalence." This notion would aid us to see if a certain implementation of a data type, t, is behaviourally isomorphic to the original specification. Thus by showing that t' is equivalent to t one may then say t' is a "correct implementation" of t.

Let Z and Z' be data structures. A *data structure homomorphism* from Z to Z' is a pair $H=(\mu,\eta)$ such that μ: $Z \to Z'$ and if $\mu(z)=z'$, with $z=(S,u)$ and $z'=(S',u')$, then η: $S \to S'$ preserving the u-relation; i.e. if $(x,y) \varepsilon u$ then $(\eta(x),\eta(y)) \varepsilon u'$. A data structure *isomorphism* is a data structure homomorphism when both μ and η are bijective.

Definition 5.2

Two data structures Z and Z' are *weakly equivalent* if there exists a data structure isomorphism from Z to Z'.

We say that two data types t and t' are *weakly equivalent* if their respective set of data structure configurations, $\bar{Z}$ and $\bar{Z}'$ respectively, are weakly equivalent.[34]

Consistent with the definition of the equivalence of twff's of the first order predicate calculus we define the equivalence of pwff's, bwff's and awff's in a likewise manner. Let ω denote any of the latter three types of formulas, then ω and ω' are *equivalent* if for any structure σ^r_{1k} that satisfies ω with s then it satisfies ω' with s and vice versa.

---

[34] Reminder: $\bar{Z}$ denotes those configurations of Z that are reachable from ϕ wrt ψ in a data type $t=(Z,\psi)$.

Theorem 5.1

Let $t=(Z_L,\psi)$ and $t'=(Z'_L,\psi')$; let p_l (b^l_ι) and p'_l (b'^l_ι) be their p-expressions (b-expressions) associated with level l of Z_L and Z'_L respectively. If for every level $l\leq L$, p_l and p'_l are equivalent, then

if $b^l_\iota \rightarrow b'^l_\iota$ and $a_l \rightarrow a'_l$ then $\bar{Z}_L \precsim \bar{Z}'_L$

$\precsim$ denotes sub-isomorphism, and $\rightarrow$ means logical implication.

Proof

Using induction on the number of nodes, let us start with the starting configurations ϕ and ϕ'. Let $\mu(\phi)=\phi'$ where μ is a morphism. Since p_l and p'_l are equivalent therefore $TCS_{L1}=TCS'_{L1}$. If for every level l, $b^l_\iota \rightarrow b'^l_\iota$, then for level L, $b^L_\iota \rightarrow b'^L_\iota$. Now, if CS_{L1}=empty then $\bar{Z}_L=\{\phi\}$; b'^L_ι, however, may be satisfied for some values of TCS'_{L1}. If this is the case, then $\bar{Z}'_L=\{\phi'\}\cup$\{other configurations\} in which case $\bar{Z}_L \precsim \bar{Z}'_L$. If b'^L_ι is not satisfied, then of course $\bar{Z}'_L=\{\phi'\}$. Once again $\bar{Z}_L \precsim \bar{Z}'_L$. Now consider the case when CS_{L1} is not empty. That is there are certain values of TCS_{L1} that $b^L_\iota(i)$ is satisfied with. But every value that b^L_ι is satisfied with, it also satisfies $b'^L_\iota(i)$. Hence $CS_{L1} \subseteq CS'_{L1}$. Assume $CS_{L1}=\{m_j: 1\leq j\leq n\}$ and $CS'_{L1}=\{m'_j: 1\leq j\leq n'\}$. Therefore those configurations having a single node and reachable from ϕ are $\iota(\phi,m_j)$, $1\leq j\leq n$. Also those configurations having a single node and reachable from ϕ' are $\iota'(\phi',m'_k)$, $1\leq k\leq n'$. Therefore for every $\iota(\phi,m_j)$ there exists a configuration $\iota'(\phi',m'_k)$ such that $m_j=m'_k$. Define μ such that $\mu(\iota(\phi,m_j))=\iota'(\phi',m'_k)$. Let us refer to the latter two

configurations, having only one node, as z_1 and z'_1, where $z'_1=\mu(z_1)$. The TCS's for each S_{1k} in a z_1 is also identical to that of some S'_{1k} in the corresponding z'_1 since p_1 and p'_1 are equivalent for every level. Now, if $b^1_\iota \rightarrow b'^1_\iota$ then for each S_{1k} in z_1 there exists a S'_{1k} in z'_1 such that $CS_{1k} \subseteq CS'_{1k}$, see the definition of CS.

Consider now two arbitrary configurations z and z', after n steps(each having n nodes), such that $\mu(z)=z'$. By precisely the same argument as above, for each S_{1k} in z there exists a S'_{1k} in z' such that $TCS_{1k}=TCS'_{1k}$; and since $b^1_\iota \rightarrow b'^1_\iota$ then $CS_{1k} \subseteq CS'_{1k}$. Therefore for every configuration having n+1 nodes reachable from z there exists a corresponding configuration reachable from z'. This correspondence is of course defined by μ as described above.

Theorem 5.2

Given two arbitrary data types t and t', there exists a procedure that stops with a YES answer if t and t' are *not* weakly equivalent.

Proof

Consider two data types $t=(Z,\psi)$ and $t'=(Z',\psi')$ where $Z=\{z : z=(S,u)\}$ and $Z'=\{z' : z'=(S',u')\}$. For the sake of brevity assume 1-level structures Z and Z'. The following proof is a construction of $\bar{Z}$ and $\bar{Z}'$ such that if t is not weakly equivalent to t' it stops with a confirmation else it

continues. In the process we try to find bijections η and μ such that η maps every S to S', where $\mu(S,u)=(S',u')$.

In order to find a bijection η which maps S to S', it must be the case that $|S|=|S'|$. Now, each $(S,u)\varepsilon\bar{Z}$ may be represented by $(S,\text{þ})$. Similarly each $(S',u')\varepsilon\bar{Z}'$ may be represented by $(S',\text{þ}')$. By lemma 2.3 and corollary 3.2, both $\bar{Z}$ and $\bar{Z}'$ are recursively enumerable. Therefore enumerate elements of $\bar{Z}$, and $\bar{Z}'$, starting from $z_\emptyset$(i.e. ϕ), and $z'_\emptyset$(or ϕ') and let $\mu(z_\emptyset)=z'_\emptyset$. Enumerate the rest of the elements $z=(S,\text{þ})\varepsilon\bar{Z}$ in the ascending order of $|S|$. If there are more than one configuration with the same number of nodes, enumerate all such configurations first, then enumerate the next set of configurations, whose number of node indices is that of the previous set plus 1. Enumerate $\bar{Z}'$ in the same fashion. Let $z_i=(S,\text{þ})$ denote a configuration of $\bar{Z}$ such that $|S_i|=i$. For every i find all the elements z_i of $\bar{Z}$ and z'_i of $\bar{Z}'$. Note that there are only a finite number of configurations with i nodes, for a data type, since the Construction Set, CS_{1k}, for every S_{1k} is finite. Let $z_i^j=(S^j,u^j)=(S^j,p)$, $0\le j\le n$, be those elements of $\bar{Z}$ such that $|S^j|=i$ for all j. In other words n is the number of configurations of $\bar{Z}$ having i nodes. Similarly let n' be the number of elements $z_i'^j=(S'^j,p')\varepsilon\bar{Z}'$ having $|S'^j|=i$, where $0\le j\le n'$. If $n\ne n'$ then stop, t and t' are not weakly equivalent; else find an η such that for every value of m and n in S^j that satisfies p, p' should also be satisfied with $\eta(m)$ and $\eta(n)$ and vice versa.

The search for such an η may require back-tracking so that if we are currently dealing with those configurations z_i and z_i' such that i=I, then the mappings η and μ that we have found so far for values of i=I-1,...,0 may no longer be valid. This implies that we have to try some other mapping, η and μ, such that it works for i=0,...,I. This strategy implies that we may have to try every possible mappings of $\{z_i : z_i \varepsilon \bar{Z} \ \& \ i \leq I\}$ to $\{z_i' : z_i' \varepsilon \bar{Z} \ \& \ i \leq I\}$; and for every such mapping try different combinations to try and find the right η. Note that the number of such mapping is finite. If such η cannot be found stop. Otherwise set i=I+1 and try to find (μ,η) as above.

For the case of L-level structures, the proof is a straight forward extension of the above.

Let (Z,O) be a data type where Z is an L-level data structure such that the CDSO set $\psi \subseteq O$, and $O-\psi$ is a, possibly empty, set which contains the auxiliary operations. Also let z_l denote a configuration of Z_l, the data structure defined at the lth level of Z.[35] Finally $\langle lkm \rangle_\lambda$ denotes a node at level λ; and $O_l \subseteq O$ denotes the set of operations at level l.

---

[35] Note that Z_l does not denote a l-level structure. It is a 1-level structure defined for level l of Z.

Definition 5.3

Let $t=(Z_L,O)$ and $t'=(Z'_{L'},O')$ be data types. A Σ-*homomorphism*, *H*, is a family of data structure homomorphisms, $H_l=(\eta_l,\mu_l)$ for $l\leq L$, preserving the operations, i.e.

$$H:\ t\rightarrow t'$$

where *H* is a family of pairs of morphisms: (μ_l,η_l) such that $\mu_l: (\bar{Z}_l,O_l)\rightarrow(\bar{Z}'_{l'},O'_{l'})$ and $\eta_l: S_l\rightarrow S'_{l'}$, $l\leq L$ and $l'\leq L'$, preserving the operations, i.e., if f and f' are functions in O and O' respectively then

$$H[\ f(z_{l_1},z_{l_2},\ldots,z_{l_n},\langle lkm\rangle_{l_q},\ldots,\langle lkm\rangle_{l_r})]=$$

$$f'(\mu_{l_1}(z_{l_1}),\ldots,\mu_{l_n}(z_{l_n}),\eta_{l_q}(\langle lkm\rangle_{l_q}),\ldots,\eta_{l_r}(\langle lkm\rangle_{l_r}))$$

Two data types $t=(Z_L,O)$ and $t'=(Z'_{L'},O')$ are *behaviourally isomorphic*, or *equivalent*, denoted by $t\backsim t'$, if (Z_L,I_UD) and $(Z'_{L'},I'_UD')$ are Σ-isomorphic and for every $z\varepsilon\bar{Z}_L$, for all l and k, if $\langle lki\rangle\varepsilon R^z_{lk}$ then there exists an $\langle l'k'i'\rangle=\eta(\langle lki\rangle)\varepsilon R^{z'}_{l'k'}$, $z'=\mu(z)$, and vice versa, where R^z_{lk} is the set of accessible nodes in z having the first and second indices l and k in common.

Let $t=(Z,O)$, and let $t'(Z',O')$ be an enrichment of t such that $\psi\subseteq O$ and $\psi'\subseteq O'$, and $O\subseteq O'$. Then t' is an *enrichment for convenience* of t if t is behaviourally isomorphic to t', otherwise t' is an *enrichment for change of behaviour* of t.

The latter may also be referred to as the implementation of t' by t.

5.3- Implementation by Emulation

Consider two almost similar structures depicted in figure 5.1. Clearly the two structures (a) and (b) are *not* isomorphic. Let us use the well known replacement theorem of predicate calculus [MAN 74] as follows: Let p_a (p_b) be the p-expression associated with the data structure represented by the structure in figure 5.1-a (5.1-b). Then

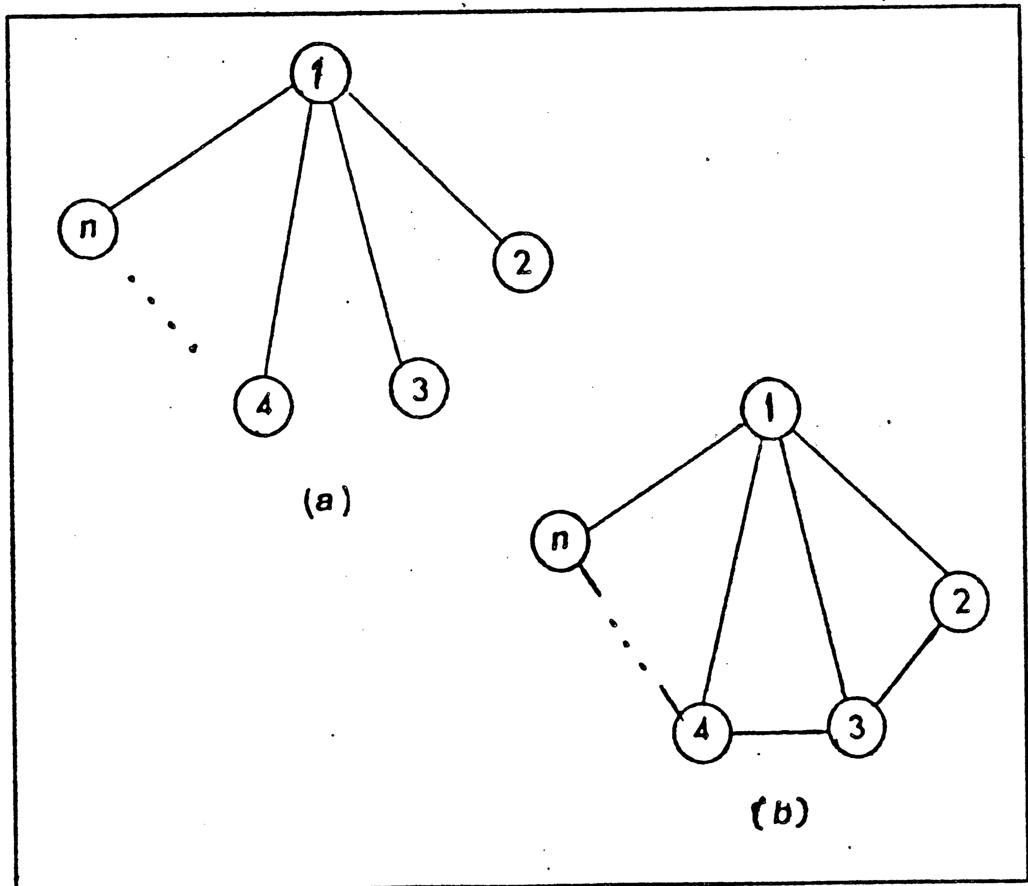


Figure 5.1.
Structural Similarity.

$p_a \equiv 1 \langle\rangle i \wedge i > 1$ and

$p_b \equiv (1 \langle\rangle i \wedge i > 1) \cup (i \langle\rangle i+1)$ or

$p_b \equiv p_a \cup (i \langle\rangle i+1)$

But the second expression on the right hand side of the last identity denotes the p-expression for a structure which is isomorphic to a list structure. Let us denote the p-expression for the latter as p_c, therefore

$$p_b = p_a \cup p_c$$

As a result a star and a list structures combined, simulate the structure of figure 5.1(b). This simple derivation illustrates the idea that, by emulating one data structure in terms of others, we can implement data-structures in terms of their equivalent structures. For behavioural isomorphism, however, we need to consider the b- and a-expressions as well as the p-expressions.

Theorem 5.3-*Correctness Theorem*

Let $t=(Z_L,O)$ and $t'=(Z'_L,O')$, then t is behaviourally isomorphic to t if for all l≤L,

$$p_l \equiv p'_l \text{ and } b_l^l \equiv b'^{\,l}_l \text{ and } a_l \equiv a'_l$$

Proof

As a corollary of theorem 5.1, because the corresponding p- and b-expressions are equivalent, $Z \simeq Z'$. That is Z and Z' are isomorphic. The behavioural isomorphism follows from the definition.

Thus it is immediate that:

$$(\text{stack}, I_U D_U P) \not\sim (\text{stack}, I_U D_U P_U\{\textit{bottomnode}\})$$

but

$$(\text{array}, I_U D_U P) \sim (\text{array}, I_U D_U P_U\{\textit{bottomnode}\})$$

It can be seen that the range of values of the Auxiliary Probe Function: *bottomnode* does not satisfy the a-expression for the **stack** for every **stack** configuration. Hence the resulting data type does not behave the same as the **stack** would.

Implementation of data types is a step by step procedure of realizing one data type in terms of the other data types. Every step would bring us to a lower level of abstraction until, finally, the lowest level of abstraction, which is the physical realization. The following theorem would enable us to ensure the correctness of this approach.

Theorem 5.4-*Implementation Theorem*

Let $t=(Z_L,O)$ be a data type with p-, b- and a-expressions at level $l \leq L$ denoted as p_l, b_l, and a_l respectively. Let t' be the data type resulted from replacing one or more of these expressions, or parts of them, at one or more levels of **t** by their equivalent expressions respectively. then **t** and **t'** are equivalent.

Proof:

Follows from theorem 5.4 and the well known replacement theorem of logic[MAN 74].

The implementation of one data type in terms of another is just a matter of "matching" their p-, b- and a-expressions. Thus complex structures may be broken down into the existing ones and therefore implemented. For example, we can realize a **stack** by introducing appropriate b-expressions and a-expressions to an **array** since their p-expressions are the same. The following examples illustrate the idea.

<u>Example</u>

Given an **array** data type with the following base expressions:

p: $i<>i+1$

b_ι: $i \geq 1$

b_δ: $i \geq 1$

a: $i \geq 1$

then,

a) A **stack** is defined by modifying the b- and a-expressions of the **array** as follows:

$b_\iota(i)$: $\forall j(j<i)$

$b_\delta(i)$: $\forall j(j \leq i)$

$a(i)$: $\forall j(j \leq i))$

b) A **binary tree** data type can be defined by modifying only the p-expression of the **array** as follows:

p: $(i<>2i) \cup (i<>2i+1)$

The b- and a-expressions remain unchanged. Figure 5.2 illustrates the node formation of the binary tree for a particular configuration of the array.

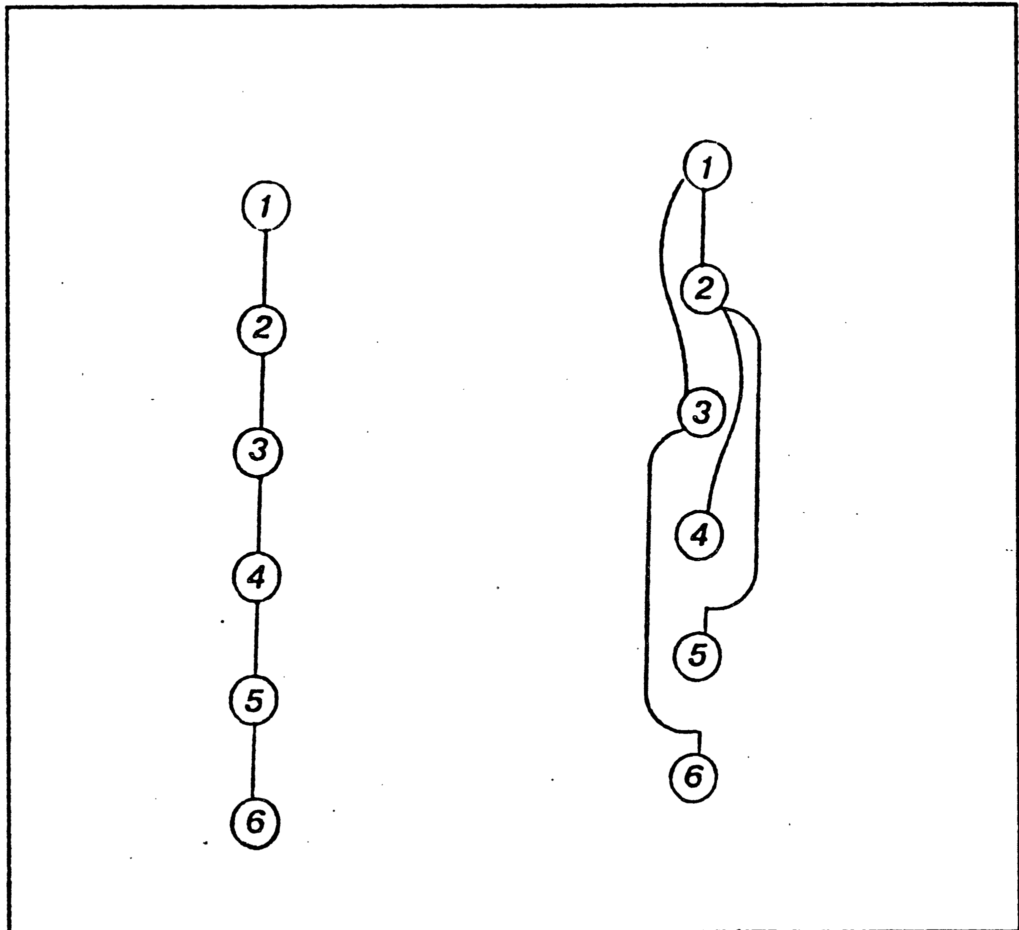


Figure 5.2
array Implementation of binary tree.

Similarly the queue, the list or any other type may be obtained by merely modifying one or more of the base

predicates of a given data type in order to arrive at the desired behaviour.

The **array** implementation of a **binary tree** presented above confronts us with an interesting problem of morphisms between the data types. It can be observed that since for a given number of nodes a **binary tree** may have more elements than an **array**, there exists no epimorphism from the **array** to the **binary tree** data type but there does exist a monomorphism. Furthermore, there exists a (partial) morphism η such that η: **tree→array**, where η is isomorphic for some subset of the the **binary tree** data type. In other words, the **array** is sub-isomorphic to the **tree**. By defining a suitable congruence relation, for example "equal-number-of-nodes", E, on the **binary tree** to get **binary tree**/E, one can define an isomorphism which maps **binary tree**/E to the **array**. The question is of course: when is a data type t' sub-isomorphic to another data type t? This may be answered immediately by the statement: when t contains t'.(Recall definition 4.1.) In general, this problem is undecidable.

Finally, it is a trivial task to enrich a **stack** in order to make it emulate the behaviour of the FIFO **queue**. The p- and a-expressions remain the same as that of **stack**. The b-expression for deletion would not be altered. The b-expression for insertion is the only change needed to simulate a **queue** behaviour. This example illustrates the extremely useful feature of "extensibility" exhibited by our

model. In comparison, in order to do the same extension in any one of the existing algebraic techniques, one has to change the whole specification. The concept of extensibility was discussed in chapter 1, it is one of the important properties desired of any specification techniques aimed at data types.

5.4- <u>Error</u>

The specification of an abstract data type is, by definition, dependent upon the operations performable on that data type. For every data structure operation, defined for a data type, there are instances of that data type that, when subjected to this operation, would result in an "undefined" state. For example for an empty stack: pop(empty stack)=undefined. Similarly push(full stack)=undefined. Ideally we are interested in specifying these undefined states so that we know what to do when such a state arises. We are also interested in knowing about the origin of the undefined state; this would help us to determine the cause of the error. For instance for a **stack of integer**, if the result of a certain operation is undefined then we would like to know which data type the undefined state is associated with- **stack** or **integer**?

The undefined states are also referred to as errors. These are no more than the $\perp$ and $\top$ that we discussed earlier. The error criteria is an integral part of every specification. Indeed in the absence of an adequate error

criteria the specification of abstract data types would have no significant value and the whole purpose of a formal specification would be defeated. The past approaches to the way the error criteria has been handled are in general either incomplete and unsuitable [GUT 75], or the error issue has become so complicated that it has proved to be a problem of unreasonable magnitude to tackle [GOG 78].

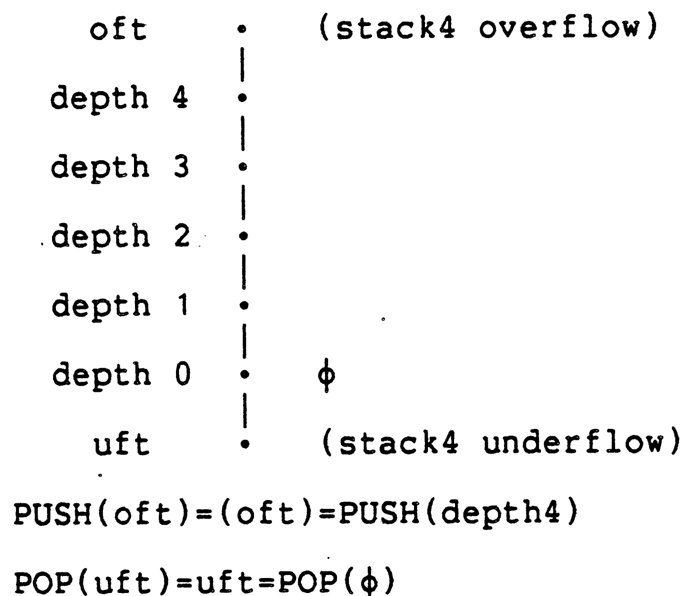
In our approach, instead of depending on extra axioms to define error instances we tackle the problem by defining the domain of the data structure operations using the appropriate p-, b- or a-expressions. In essence, when an operation results in error it causes a transition from one reachability equivalence class to another. For instance, consider a **stack10**, which is a **stack** of maximum depth 10. The 11th push causes an error instance. Such an error instance may be interpreted either as the "overdefined" element of the **stack10**, namely $\top$, or we may interpret the 11th push operation as one causing a transition from **stack10** data type to another, e.g. **stack20** or **stack50**. The latter approach has a number of shortcomings. For instance, push or any other DSO defined for **stack10** can only result in elements of **stack10**, so how can it possibly result in elements of **stack50** or any other type? Even if we did overcome the latter problem it would not be clear which particular type the transition is made into, namely **stack11** or **stack67**? The former problem, however, is more significant and profound than the latter. It leads to the

same difficulties encountered by some advocates of the algebraic technique in their error treatment [GUT 75].

We shall pursue the idea that the $\perp$ and $\top$ elements of a data type define the error instances of that data type. Recall that for every data type there are two fixed points with respect to all constructor operations of that data type; these are oft and uft whose values are represented by $\top$ and $\perp$ respectively. Thus every DSO is strict or bottom preserving. Since the $\top$ and $\perp$ define the "limits" or "boundaries" of a data type, it is then possible to define such boundaries using the b-expressions. This is indeed what the b-expressions were primarily intended to do.

Example

A **stack4** lattice is illustrated below.



PUSH(oft)=(oft)=PUSH(depth4)

POP(uft)=uft=POP(ϕ)

In fact theorem 3.1 would ensure that all the "non error" configurations are reachable from each other and all

the error configurations are trapped in their own equivalence classes. Hence no error configuration may reach a non error configuration and vice versa.

5.4.1- Domain of DSO's

In general, we are interested to know precisely what are the domain of DSO's. It is sufficient to consider only the operations of the characteristic DSO set ψ since all other operations are defined in terms of these.

Recall the definition of i- and d-functions in chapter 3; their domain was described to be $\mathbf{Z} \times \mathbf{L} \times \mathbf{K} \times \mathbf{M}$, where **L**, **K**, **M** are some subsets of $\mathbf{N}^+$. For a given level of a data type, we are only concerned with **Z** and **M**, keeping $l \varepsilon \mathbf{L}$ and $k \varepsilon \mathbf{K}$ at a fixed value. Since for given values of l and k, the only variable is m and it is precisely this variable whose range of values affects the behaviour of the structure. Similarly, for the case of the probe functions it suffices to consider $\mathbf{Z} \times \mathbf{M}$ as their domain.[36] The problem is how to define the appropriate subsets of **Z** and the set **M** for each kind of function i-, d-, and p- respectively. As an example consider the above **stack4** data type. Let Li denote a **stack** configuration of depth i, then

$\bar{\mathbf{Z}} = \{\phi, L1, L2, L3, L4\}$

---

[36] Note that the probe function is really a mapping of the form $z \times \langle lkm \rangle \rightarrow \langle lkm' \rangle$; but for fixed values of l and k every <lkm> may be denoted by, merely, $m \varepsilon \mathbf{M}$.

M- varies dynamically depending on the depth of the stack, i.e. it depends on $z \varepsilon Z$. We saw that this was determined by the b-expression for the stack.

For a **stack** configuration of depth 3, $M=\{S_{114}\}$. Since $s[i]=S_{114}$ is the only element of the TCS of L3 (a configuration of depth 3) that **stack** satisfies $b_\iota(i)$ with. Consider now a stack of depth 4, L4. The value of **M** would be equal to { }, which implies that nothing may be pushed on the stack. Note that the elements of **M** in the above example are determined by the b-expressions of the **stack**. Hence for stack of depth 4, **M** is empty, and therefore no more nodes may be pushed onto it. As a result an error state is never entered. The task of introducing extra constraints in order to alter the construction and destruction sets is a rather straight forward task. For example, the depth of the **stack** configurations may be limited to a maximum size of 100 by ANDing the extra constraint of $i \leq 100$ to its b-expression, b_ι.

It should now be obvious that for an i-function ι: $Z \times L \times K \times M \rightarrow Z$, where $\iota(z,l,k,m)=z'$, $M=CS_{lk}$. Similarly for a d-function δ: $Z \times L \times K \times M \rightarrow Z$, where $\delta(z,l,k,m)=z'$, $M=DS_{lk}$. But we have already seen that the CS's and the DS's for a given $z \varepsilon Z$ are recursive sets. Hence for every configuration $z \varepsilon Z$ the domain of the i- and d-functions are recursive at every level, for every structure (S_{lk},u_{lk}).

It can be seen that the process of specification is reasonably straight forward. The "extension" of specifications is even simpler. Behavioral changes may be made through "addition" or "omission" of appropriate constraints in one or more of the predicate expressions. Hence given a data type with its related b-expressions it is easy to change such expressions in order to come up with a new desired behaviour. The same approach may indeed be adopted in order to modify the behaviour of a data type via p- and a-expressions. Such facility makes the process of implementation or extension a trivial task. For instance, given the b-expressions for the **stack**, we could readily specify (or implement) a FIFO **queue** by making minimal changes to the **stack** b-expression for insertion. The a- and p-expressions remain unchanged. The same process, using the algebraic approach, means that we will have to discard the "gifted" **stack** specification and start with a fresh specification of the **queue** and yet worry about the correctness of this latter specification.

CHAPTER VI

CONCLUSIONS AND FURTHER WORK

The scheme which we have adopted, for specification of data types, is a blend of definitional and operational methodologies. The advocates of the axiomatic techniques reject the graph models because of their implementation bias. The degree of such a bias varies from one graph model to another. We must emphasize that the Σ-structures are not to be interpreted as actual memory layout of data nor do the edges represent access paths as in Earley's approach [EAR 73]. In our approach, a much more abstract notion of access is employed, namely the a-expressions. In general the presence of implementation bias may be detected in almost every technique; this includes both the explicit as well as implicit techniques. A problem of considerable importance is the difficulty of construction and comprehension which faces the present formal implicit techniques. Our usage of graph structures has considerably alleviated this problem.

A major problem with previous techniques has been to define a set of operations to completely and consistently characterize the behaviour of a data type. The characteristic set ψ and the properties required of them

provide us with sufficient information to characterize data types both completely and consistently. By employing the DSO functions, viz. the i- and d- functions, we showed that data types may be modelled as lattices.

As a valuable merit to our approach, we showed that all that is required to specify a data type is to specify the p-expressions, b-expressions and the a-expressions for the desired data type. Also demonstrated was the fact that a given data type specification may conveniently be extended to exhibit a different behaviour. This was exemplified in the case of the **stack** where by making a small change in a single expression the FIFO **queue** was realized. Because of this built-in facility of extensibility, in our model, in most cases one can realize the desired data type by a few minor and simple alterations to an existing specification. However, as noted earlier in the introduction, the price that one pays for such conveniences is the minimality of the specification. For example, the **set** data type may only be expressed as an **indexed set**. Thus an extra constraint is unnecessarily imposed on the **set** data type. Among other potentials of the base predicate expressions is their use in program verification. They provide the user with predicates that must hold for every instance (configuration) of a data structure so that every configuration is true in the sense that was defined in chapter 4. Furthermore, both the soundness and completeness properties of a specification

were demonstrated so that all the true configurations are deducible, and all the deducible configurations are true.

As a further facility to specify data types of a more complex nature we introduced type manipulation operations. These operations enable us to arrive at more complex data types without the burden of redefining their operation set; since their characteristic set is automatically obtained from the constituent parts. This is indeed a very convenient tool in specifying less commonly used data types and be certain of their validity. Thus, data types may be defined independently from each other, and then combined according to the users' recipe. This philosophy is highly related to the concept of abstraction of data found in SIMULA [DAH 66] and other languages that followed from it. It is clear that parametrization of data types is a direct application of the embed TMO.

Two types of enrichment were introduced: enrichment for convenience and enrichment for change of behaviour. Using the inherent properties of our model, it was shown that, one may enrich a data type by any other operation. Such auxiliary operations are defined in terms of the more primitive operations. The "correctness" of the former is automatically decided by the elements of the existing base predicate set. We also demonstrated that in order to enrich a data type for change of behaviour we must change its base

predicates. This process was referred to as implementation by emulation.

The base predicate expressions are of tremendous value in order to prevent reaching the error instances of a data type. These expressions are employed to define the domain of the characteristic functions of every data type. This idea alleviates the complex and tedious error theory approach of the past. By merely specifying the base predicate expressions of a data type the domain of its characteristic functions, as well as the error instances, would become decidable. Hence the burden of specifying the extra "error equations" [GOG 78] and other exceptional equations such as that of [KAP 80] are removed.

An important and valuable merit of our approach is the fact that the issue of parallelism of operations is inherently built into the specification of a data type. The b-expression for insertion (say) of a data type defines the construction set of any configuration of that data type. The latter set would in turn define the nodes which may be inserted into the configuration. As a result of this arrangement, for every given configuration the elements of the construction set may all be inserted in parallel into that configuration. Likewise the elements of the destruction set may all be deleted in parallel. Toward this end, parallel insertion and/or deletion and/or probe operations may be defined and in addition such operations

may be used to define other operations of a more complex nature. With this facility, a number of operations may be carried out at different levels of the structure on possibly distinct nodes of that structure concurrently. The correctness of such operations is guaranteed by the base predicate expressions. The ability to define operations acting concurrently on the same set of data is indeed invaluable in many systems such as data bases and operating systems.

The combination of structure and behaviour enables us to specify certain criteria which have not been demonstrated before by other techniques. For example the a-expressions,or equivalently the probe function, and its use in security of data. For example: in an operating system environment one may readily allocate different a-expressions to different users, for the same data type, so that the more priviledged users have more access than others. Finally by renumbering the nodes it is possible to reconfigure data types such that a given data type may exhibit a characteristic distinct to that originally intended. Thus a set of data items may perform different behaviours to different users, all at the same time, depending on the numbering scheme associated with its nodes. This, of course, is a very useful tool in data base systems. Appendix B presents a brief introduction on the potentials of our model regarding this concept.

<u>Further Work</u>

- Extension of our model to specification of "algorithms." An *algorithm* is a collection of one or more data types. To define an algorithm, we would like to bring together, or "combine", all the necessary elements of different algebras, and define operations which may or may not cause transitions between different sorts of the resulting algebra. For example, in order to sort elements of the **array of integers**, we may need the help of the **bool** data type. In the presence of **bool**, we are able to define an operation such as >: $\mathbf{N \times N \rightarrow bool}$. Other operations may also be added such as EX(z,i,j) operation which exchanges the contents of the nodes i and j of the array configuration z.

- In some cases where two data types t_1 and t_2 are combined to get $t_1 \times t_2$, we may be interested in adding extra constraints to the resulting data type. We may be interested in modifying the b-expression, i.e. enrichment for change of behaviour; or it may be necessary to add extra axioms in order to define some desired equivalence relation on the set of data structure configurations, Z. As an example consider two terms of a **set-of-char** data type: t_1 and t_2. If t_1 and t_2 contain the same elements then they are equivalent no matter what is the order in which the

elements are inserted. The following rule of reduction may be introduced for the **set-of-char:** let sε**set-of-char** and a,bε**char**, then

$$\iota(\iota(s,i,a),j,b)=\iota(\iota(s,i,b),j,a)$$

- Specification of parallel operations.

- Possible extension of the model such that each node at a given level may contain (behaviourally) different structures. In the presence of such *(extended)* Σ-structures, we can also define a "join" TMO to combine data types.

APPENDIX A

Examples

A few examples of data type specification is presented in this appendix. More specifications can be obtained either by direct specification or by the use of TMO's. We also demonstrate specification of an algorithm, namely Binary Search Tree Insertion, using our model.

array

p: $i<>i+1$

$b_\iota(i)$: $i \geq 1$

$b_\delta(i)$: $i \geq 1$

$a(i)$: $i \geq 1$

stack

p: $i<>i+1$

$b_\iota(i)$: $\forall j(j<i)$

$b_\delta(i)$: $\forall j(j \leq i)$

$a(i)$: $\forall j(j \leq i)$

binary tree

p: $(i<>2i) \cup (i<>2i+1)$

$b_\iota(i)$: $i \geq 1$

$b_\delta(i)$: $i \geq 1$

$a(i)$: $i \geq 1$

int

There are two i-functions $i_\emptyset$ and i. The b-expressions are: for $z \varepsilon int$

$b_{\iota_\emptyset}(z)=$*true* if $z=\phi$

=*false* otherwise

$b_\iota(z)=$*true* if $z\neq\phi$

=*false* otherwise

$b_\delta(z)=$*true* if $z\neq\phi$

=*false* otherwise.

i is of course the Peano's suc function, where d is the pred function.

nat

The i-functions are the same as int with the following added restriction. The b-expressions are:

$b_{\iota_\emptyset}(z)=$*true* if $z=\phi$

=*false* otherwise

$b_\iota(z)=$*true* if $z\neq\phi$

=*false* otherwise

$b_\delta(z)=$*true* if $z\neq\phi$ or $z\neq\iota_\emptyset(\phi)$

=*false* otherwise

bool

There are two i-functions i_1 and i_2 and their inverse. For both i-functions i_j, $1\leq j\leq 2$,

$b_{\iota_j}(z)=$*true* if $z=\phi$

=*false* otherwise.

$b_\delta(z)$=*true* if $z \neq \phi$

=*false* otherwise

We may also enrich **bool** to include functions other than the insertion and deletion functions. For example the AND function is described as follows:

Let b_1, $b_2 \varepsilon$**bool**

AND(b_1, b_2)= *if* $b_1=i_1(\phi)$ or $b_2=i_1(\phi)$

then $i_1(\phi)$

else $i_2(\phi)$.

<u>string</u>

There are 26 constructors, each having a b-expression as follows: let zε**string**

$b_{\iota_j}(z)$= true if $z \neq \phi$ $1 \leq j \leq 26$

= false otherwise.

$b_{\iota_\emptyset}(z)$=true if $z=\phi$

=false otherwise

$b_\delta(z)$= true if $z \neq \phi$

= false otherwise.

<u>balanced tree</u>

A **balanced tree** data type is one that contains binary tree elements such that every node of the latter is balanced. Where balancing of a node j means the difference

between the left and right subtree of j is less than or equal to 1. The specification is an extension of the **binary tree** specification.[37]

p: $i<>2i \cup i<>2i+1$

b_ι: $i\geq 1 \cup \forall j(((1\geq j\geq \downarrow i/k\downarrow) \wedge k=2^r \wedge r\varepsilon N^+)\rightarrow |BF(j)|\leq 1)$

b_δ: $i\geq 1 \cup \forall j(((1\geq j\geq \downarrow i/k\downarrow) \wedge k=2^r \wedge r\varepsilon N^+)\rightarrow |BF(j)|\leq 1)$

a: $i\geq 1$

where BF(j) is a function which returns the Balancing Factor of j defined as follows: Define the height of a node j as

$$h(j)=\max(h(2j),h(2j+1))+1 \quad \text{if } \exists k\ p(k,j) \cup j=1$$
$$= -1 \quad \text{otherwise.}$$

then, $BF(j)=Z(h(2j))-Z(h(2j+1))$

where $Z(i)=i$ if $i\geq 0$

$=0$ otherwise.

traversable stack

An example of a **"traversable stack"**(**T.stack**) was cited in [MAJ 77A], and it was demonstrated that such a simple and plausible data type is not axiomatizable using the algebraic approach. In here, we shall show how simple it is to arrive at the desired specification using our approach.

It is required that the **T.stack** can perform the following operations:

pushL- same as the conventional **stack** push.

---

[37] $\downarrow x\downarrow$ means the Floor value of x.

popL- same as the conventional stack pop.

readL(z,i)- probe a node indexed i.

downL(z,i)- returns the node index neighbouring to i in the direction of the bottom of the stack.

returnL(z,i)- returns the top node of the stack.

Such a behaviour may be obtained using the following base predicate set:

p: $i<>i+1$

$b_\iota(i)$: $\forall j(j<i)$

$b_\delta(i)$: $\forall j(j\leq i)$

a(i): $i\geq 1$

Note that the above behaviour is obtained by a simple adjustment to an already existing **stack** specification. This facility of extensibility, in our model, is most useful in such cases where an almost similar behaviour to an existing data type is needed. Now we may introduce some auxiliary operations in order to implement the desired operations: downL, returnL etc. The operation downL may be defined as:

downL(z,i)=i-1

Since the range of downL is the set of node indices, it is therefore a probe function. For downL to be a legal probe function, for every i, $i-1\in\{j: a(j)\}$. For example, let z be a stack of length 1; then $downL(z,1)=1-1=0\notin\{j: a(j)\}$; therefore downL(z,1)=error since a(0) is not satisfied.

Finally, the operation returnL(z,i) may be readily defined. This is the same as the p-function of **stack** introduced earlier, i.e.

$$p(z,i)=\lambda i.\ \forall j(i \geq j) \rightarrow i,\ p(z,i+1)$$

As we mentioned earlier, to specify the above **T.stack** with the algebraic approach, an infinite number of equations may be needed. This is not of course acceptable. To get around this an extra operation is needed, "hidden" to the user, such that the **T.stack** may be axiomatized in a finite number of steps. A pointer, viz. shove, which is always pointing at the bottom of the stack, is used[TWW 79]. This would enable one to detect when and if the bottom of the stack is reached. It should be noted that the introduction of shove presents us with some information which is not really needed for a minimal specification, i.e. an implementation bias. Furthermore, the specification given in [TWW 79] does not include error instances, i.e. the so called error equations. To do so it would be a formidable task hardly worth the effort.[38]

---

[38] A potential use of **T.stack** would be as a *syntax analyzer*.

APPENDIX B

Dynamic Change of Behavior

Different "views" and "authorization" of data are crucial issues in data base design and maintenance[CHA 75]. The potential of b-expressions to define different views of the same data for different users is apparent. This is accomplished by having different b-expressions and a-expressions for each user. Furthermore, a more global change of behaviour may be observed if the p-expressions are modified, by the Data Base Adminstrator(say), in order to allocate distinct views to each user of the data base. This may be done by adding more, or less, constraints to the b-, p- and a-expressions already specified. Consider an **array** of some data items, by having different b- and a-expressions for different users distinct behaviours would be exhibited by the data items. Thus for the given **array**, each user would be concerned only with his own "view", i.e. his own b- and a-expressions. Another, more subtle, way of maintaining different views is to keep the a-, b- and p-expressions constant and change the numbering associated with the nodes. In the presence of a clever algorithm very efficient changes of behaviour may be observed at the cost of permuting the node numberings.

Maintaining different views of data for different users, concurrently, is most efficient since it saves duplication of data items into behaviourally different structures. Change of b- and a-expressions only causes behavioural reshuffling while maintaining the same underlying structure. A structural as well as behavioural reorganization may result by either a "renumbering" of the nodes of the structure or changing the p-expressions associated with the structure. Consider, for example, the problem of maintaining an **ordered list** of items. This problem can be handled by constantly updating the renumbering of the nodes dynamically as the nodes are inserted. So that the nodes would be numbered in ascending (or descending) order depending on their content. Another view of the same **ordered list** may be held by numbering the nodes, not according to their alphabetic ordering but by some other "key" value. Hence depending on the type and frequency of the queries different numbering of the nodes may be allocated, or varying p-expressions may be adopted to imply variable relationships between the items.

APPENDIX C

λ-Notation-Basic Idea

Suppose $\mathbf{E}(x)$ is some expression involving x such that whenever $d \varepsilon D$ is substituted for x- and we shall denote the result of such a substitution by $\mathbf{E}(d)$- the resulting expression (viz. $\mathbf{E}(d)$) denotes a member of D'. For example if both D and D' are set of natural numbers then $\mathbf{E}(x)$ could be $x+1$ (so $\mathbf{E}(5)=5+1=6$) or perhaps $x \times x$ (then $\mathbf{E}(5)=5 \times 5=25$). For such expressions the notation

$$\lambda x.\ \mathbf{E}(x)$$

denotes the function $f\colon\ D \rightarrow D'$ such that:

for all $d \varepsilon D$ then $f(d)=\mathbf{E}(d)$

For example:

i) $\lambda x.\ x+1$ denotes the successor function.

ii) $\lambda x.\ x \times x$ denotes the squaring function.

iii) $\lambda x.\ (x=0) \rightarrow$ *true*, *false* denotes test-for-zero function.

An expression of the form $\lambda x.\ e(x)$ is called a λ-expression, x is its bound variable and $e(x)$ its body.

This is the central idea of λ-notation. For more elaboration on this topic the reader is referred to [CHU 56,

KAT 48, GOR 79]. Note that the body of a λ-expression always extends as far to the right as possible; thus $\lambda x.x+1$ is $\lambda x.(x+1)$ not $(\lambda x.x)+1$.

APPENDIX D

More on Insertion and Deletion

The characteristic set of DSO's requires inverse operations for every i- and d- functions such that terms can be reduced to their minimal form. Consider a term $z=i_1(d_1(i_1(i_1(d_1(i_1(\phi,1),1),1),2),2),2)$; then the question is whether we can reduce z to $z=i_1(i_1(\phi,1),2)$. Indeed this can be done by the reduction rule given in chapter 3. If the node indices match, clearly the reduction can be performed no matter what sequence of i- and d-functions exists provided that the p- and b-expressions are satisfied. There are cases however where the node to be deleted has a non-empty successor set. For example consider an array configuration, z, of length 5. The node indexed 3(say) may be deleted by the appropriate d-function d as follows:

$$d(i(i(i(i(i(\phi,1),2),3),4),5),3)$$

As a result we end up with a "gap" caused by the removal of a node indexed 3. What happens to nodes 4 and 5? They cannot exist without the presence of their predecessor node 3. In fact the resulting configuration is an element of the UFT. There are a number of alternatives as to what we should do, with the successors of a deleted node, such that the resulting term is not an error term. One alternative is

to make this decision at the level of the primitive d-function. This is discussed below, giving the user the choice to either define an alternative host for the successor set of the deleted node or delete the successor nodes all together. Same philosophy is pursued when an existing node is displaced by a new insertion.

A more general approach, however, is to allow the user to specify the "destiny" of the successor set by means of some "auxiliary equations." The auxiliary equations would enable us to define functions in terms of the existing ones. In the above example one can use such a tool to define the function d' such that d' not only deletes node 3 but also specifies whether node 4 and 5 should remain in the structure, and if so what index values should they take. For example after deletion of node 3, node 4 could change to 3, and node 5 may be changed to 4. Alternatively both 4 and 5 may be deleted, i.e. their index is changed to Ø. Let us assume that we are interested in the former approach of shifting the node indices down by 1. Then d' should be composed of the following sequence of primitive operations; assume z is the original array configuration of length 5 described above:

$$z'=i(i(d(d(d(z,5),4),3),3),4)$$

It can be seen that by our reduction rule the above term reduces to:

$$z'=i(i(i(i(\phi,1),2),3),4)$$

Note that in this example, since the array nodes contain empty structures, the resulting configuration z' would be the same no matter which one of the 5 nodes is deleted provided that the successor nodes are "shifted down" by one as assumed above.

Constructors with Hidden Parameters

The i- and d-functions defined in here are intended as an alternative to those defined in chapter 3. They may be employed in cases where the definitions 3.2 and 3.3 are either not acceptable or not convenient to use. An example of this situation would be the FIFO **queue** data type, where deletion always requires "displacement" of the nodes already in the queue. Note that in this case the deletion function may just as well be defined in terms of the definitions 3.2 and 3.3.

Definition 3.2'

Let $z=(S,u)$, $z'=(S',u)\varepsilon Z_L$, where Z_L is a data structure and let b^1_ι be the b-expression associated with level 1, $1 \leq l \leq L$. Also let η denote an isomorphic mapping of two Σ-structures. An *insertion function* (i-function) is a partial mapping:

$$\iota \quad : Z_L \times N^4 \to Z_L$$

such that, $\iota(z, l, k, m, n)=z'=(S', u')$ where $1 \leq l \leq L$, and the following conditions hold at level l:[39]

---

[39] This equation denotes: insert a node indexed m at level l of the configuration z in node k of the (l+1)th

1) $b_l^1(S_{lkm})$

2) If $l<L$ then $\sigma_{l+1,q}$ satisfies $\alpha_{l+1}(S_{l+1,q,k})$ for some q.

3) $S_l' = \{S_{lkm}'\} \cup$

$\{S_{lkj}' : E(S_l, S_{lkj}) \wedge j \notin sucset(m)\} \cup$

$\{\eta(S_{lki})' : (S_{lki} \varepsilon sucset(S_{lkm} \cup i=m) \wedge b_l^1(i)\}$

- such that $(S_{lkn}', \eta(S_{lkm})'') \varepsilon u_l'$ and $\eta(S_{lkm}) \notin sucset(S_{lkn})$ and $(sucset(\eta(S_{lkm})'), p_l) \approx (sucset(S_{lkm}), p_l)$.

4) The m-index of the lth level nodes contained in nodes (k+1), (k+2),...of the (l+1)th level will change only if the maximum value of the m-indices in node k of (l+1)th level is modified. The renumbering follows from the numbering rules of Σ-structures described in definition 2.2. Also the k-index of the (l-1)th level nodes contained in m and the sucset(m) nodes in k, k+1, k+2,...(if any), would change correspondingly according to definition 2.2. The indexing of the nodes $S_{\lambda\kappa\mu}$, where $\lambda=l+1$ or $\lambda \geq l$ and $\kappa \leq k-1$, would remain the same such that for every $S_{\lambda\kappa\mu} \varepsilon S$ which satisfy the latter condition, there exists a node $S_{\lambda\kappa\mu}' \varepsilon S'$.

Definition 3.3'

A *deletion-function* (d-function) is a partial mapping: (following the same notation as in definition 3.2)

---

level. The displaced sucset(m), if any, becomes the sucset(n) such that the node previously labeled S_{lkm} would become $\eta(S_{lkm})'$, for some η. The resulting configuration is z'; and the inserted node is labeled S_{lkm}'. Note that, the primed sets and symbols refer to the new configuration.

$$\delta : \quad Z_L \times N^4 \rightarrow Z_L$$

such that $\delta(z,l,k,m,n)=z'=(S', u')$, and the following conditions are satisfied:[40]

1) $b_\delta^l(S_{lkm})$
2) If $l<L$ then $\sigma_{l+1,q}$ satisfies $\alpha_{l+1}(S_{l+1,q,k})$ for some q.
3) $S_l' = \{S_{lkj}' : j \notin sucset(m) \wedge j \neq m\} \cup$
   $\{\eta(S_{lki})' : S_{lki} \in sucset(S_{lkm}) \wedge b_\delta(i)\}$
   - such that the latter set is a subset of sucset(S_{lkn}') and structurally isomorphic to sucset(S_{lkm}).

If n=0, then the elements of sucset(m) will become empty nodes, hence they would be deleted.

4) The m-indices, of the l th level nodes, contained in (k+1), (k+2),.... nodes of the (l+1)st level, if any would change if the maximum node index in the kth node of (l+1)st changes, according to the definition 2.2. Also The k-index of the l-1 level nodes and the levels below will change correspondingly. The nodes $S_{\lambda\kappa\mu}$, with λ, κ and μ the same as that of definition 3.2, remain unchanged.

It should be noted that the extra parameter n in the above functions are really "hidden" parameters. Therefore to a user such functions would behave the same way as those

---

[40] The deletion equation denotes: delete the node at level l of configuration z,indexed m, in node k of the (l+1)st level. The displaced sucset(m),if any, will become the sucset(n). The resulting configuration is z'.

of definitions 3.2 and 3.3. For a queue, say, the value of n in the deletion function of 3.3' is always equal to 1

BIBLIOGRAPHY

BIBLIOGRAPHY

[ARB-75] Arbib, M. H.; E.G. Manes,: **Arrows, Structures and Functors**, Academic Press inc. 1975.

[AFS-80] Afshar-Khajevand, S.: **Towards a Behavioral Study of Data Structures**,Ph.D. Thesis, CICE, Univ. of Michigan, Ann Arbor MI. 1980.

[BRO-80] Brodie,M.L.: **Data Abstraction for designing Database-intensive Applications**,ACM,SIGPLAN Notices, 16, 1, January 1981.

[BUR-77] Burstall, R.M.; J.A. Goguen,: **Putting Theories together to Make Specification.**,5th intl. conf. on AI, pp. 1045-1058, 1977.

[CHA-75] Chamberlin, D.D. et al,: **Views, Authorization, and Locking in a Relational Data Base System**, AFIP Natl. Comp. Conf, vol 44 pp425, 1975.

[CHU-36] Church, A.: **Some Properties of Conversion**, Princeton University, Princeton NJ.; 1935.

[CHU-56] Church, A.: **Introduction to Mathematical Logic**,vol 1, QA-9.c55, 1956.

[DAH-66] Dahl O.J. Nygaard, K.: **Simula-An Algol-based Simulation Language**, CACM 9, Sept. 1966, pp.671-678.

[DIM-69] D'Imperio, M.: **Data Structures and Their Representations in Storage.**,Annual Review in Automatic Programming, S, 1969.

[EAR-73] Earley, J.: **Relational Level Data Structures for Programming Languages**,Acta Informatica 2, pp293-309, 1973.

[END-72] Enderton, H. B.: **A Mathematical Introduction to Logic**, Academic Press inc. 1972.

[GHM-76] Guttag, J.V. et. al.: **Abstract Data Types and Software Validation**,Information Sc. Inst. Rep. RR-76-48; Marina del Rey, Cal. 1976.

[GOR-79] Gordon, M.J.C.: **The Denotational Description of Programming Languages-An Introduction**, Springer-Verlag, New York, 1979.

[GOG-75] Goguen, J.A. et al,: **Abstract Data Types as Initial Algebras and Correctness of Data Representation**,Proc. Conf. Comp. Graphics, Pattern Recognition and Data Structure, 1975.

[GOG-78] Goguen, J.A. et al,: **An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types**,Current Trends in Prog. Methodology IV, ed. R. Yeh Prentice-Hall 1978.

[GUT-75] Guttag, J.V.: **The Specification and Application to Programming of Abstract Data Types**,Ph.D Thesis, Computer Systems Research Group Tech. Rep. CSRG-59, Dept. of Com. SC. U. of Toronto, 1975.

[GUT-77] Guttag, J.V.: **Abstract Data Types and the Development of Data Structures**,CACM, 20, 6, June 1977.

[GUT-78] Guttag, J.V., J.J. Horning: **The Algebraic Specification of Abstract Data Types**, Acta Informatica 10,27-52; 1978.

[HAN-69] Hansen, W.J.: **Compact List Representations: Definition, Garbage Collection, and System Implementation**,CACM, 12, 9, Sept. 1969.

[KAP-80] Kapur, D.: **Towards A Theory for Abstract Data Types**, Ph.D. Thesis MIT, Cambridge, Mass. 1980.

[KAR-75] Karp, R.M. et al,: **Near Optimal Solutions to a Two Dimensional Placement Problem.** SIAM J. Comp. 4, 271-286, 1975.

[KAT-48] Kattsof, L. O.: **A Philosophy of Mathematics**, The Iowa State College Press, Ames, Iowa. 1948.

[LIS-74] Liskov, B. Zilles S.: **Programming with Abstract Data Types**,SIGPLAN Notices 9, 4, Apr. 1974.

[LIZ-77] Liskov, B. Zilles, S.: **An Introduction to Formal Specifications of Data Abstractions**, Current Trends in Prog. Methodology, IV ed. Yeh, Prentice-Hall 1978.

[MAJ-77A] Majster, M.E.: **Limits of the Algebraic Specification of Abstract Data Types**,ACM, SIGPLAN Notices, 12,10,Oct. 1972.

[MAJ-77B] Majster, M.E.: **Extended Directed Graphs, a Formalism for Structured Data and Data Structures**,Acta Informatica 8, pp. 37-59, 1977.

[MAN-74] Manna, Z.: **Mathematical Theory of Computation**,McGraw Hill, 1974.

[MOI-80] Moitra, A.: **A Note on Algebraic Specification of Binary Trees**,ACM SIGPLAN Notices, 15, 6, June 1980.

[NOU-79] Nourani, F.: **Abstract Implementations and Their Correctness**, SEL, CICE Program ECE Department-U. of M. Ann Arbor-Submitted for Publication 1979.

[RAN-72] Randall, L.S.: **A Relational Model of Data for the Determination of Optimum Computer Storage Structures**,Rome Air Dev. Ctr. Tech. Rep. RADC-TR-72-25, Feb. 1972.

[ROS-70] Rosen, B.: **Tree Manipulation Systems and Church-Rosser Theorems**,Harvard Univ. Cambridge, Mass. 1970.

[ROS-71] Rosenberg, A.L.: **Data Graphs and Addressing Schemes.**,JCSS5, pp. 193-238 1971

[ROS-78] Rosenberg, A.L.: **Storage Mappings for Extendible Arrays**,Current Trends in Programming Methodology IV, ed. R. Yeh, Prentice-Hall 1978.

[ROS-81] Rosenberg, A.L.: **On Uniformly Inserting One Data Structure into Another**, CACM, 24, 2, Feb. 1981.

[SET-74] Sethi, R.: **Testing for the Church-Rosser Property**,J. of ACM, vol 21, 4, 1974.

[SHN-74] Shneiderman, B.: **Structured Data Structures**, CACM, 17, 10, Oct. 1974.

[STO-77] Stoy J. E.: **Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory**, MIT press, 1977.

[TAR-66] Tarski, A.: **Equational Logic and Equational Theories of Algebra**,Contributions to

Math. Logic, Proc. Logic Colloquium, Hanover 1966.

[TWW-79] Thatcher, J.W. et. al.: **Data Type Specification: Parameterization and the Power of Specification Techniques**, Math. Sc. Dept. T.J. Watson, RC.RC7757 Yorktown Heights, NY; 1979.

[WON-75] Wong, C.K. Maddocks, T.W.: **A Generalized Pascal's Triangle**, Fibonacci Quarterly, 13 pp. 134-136, 1975.

[ZIL-74] Zilles, S.N.: **Algebraic Specification of Data Types.**, Project MAC Prog. Rep. 11, MIT Cambridge, Mass. 1974.

[ZIL-75] Zilles, S.N.: **An Introduction to Data Algebras**, IBM Research Laboratory, San Jose, Calif. 1975.

UNIVERSITY OF MICHIGAN
3 9015 03026 9677