MODELLING STRUCTURES FORMALISM

Lawrence A. Rowe

This work was partially supported by the National Science Foundation under Grant GJ-1045.

TECHNICAL REPORT #52 - November 1974

Acknowledgement

I would like to take this opportunity to acknowledge my thesis advisor Fred Tonge for his valuable contributions to this work and for his constant guidance during my graduate studies. I also want to thank Dave Farber for providing an environment in which to pursue these research activities.

Introduction

This paper presents a formalism for modelling structures. The term modelling structures is used in the sense suggested by D´imperio [1], namely, a modelling structure is an abstract, psychological, arrangement or description of information used in human problem-solving. The concept of modelling structure is not restricted to problem-solving using a computer. Data structures, on the other hand, has traditionally referred to the representation of information used in problem-solving involving a computer. Unfortunately, depending on the domain of discourse, the phrase data structures has many different meanings, including, machine independent data representations, language dependent data types, and implementation dependent data representaions. For this reason, and because we are interested in how information associated with a problem is organized in order to express an algorithm, the formalism presented deals with the general problem of information representation, independent of the way it is mapped into a physical storage medium. Modelling structures are used, explicitly or implicitly, in a program written in some programming language to implement an algorithm. Thus, the formalism is concerned with how relations among objects can be represented and operated upon.

A modelling structure is an abstraction of structural relations among a collection of objects, called the elements of the structure, and constraints on the operations that can be performed on the structure and its elements. This notion is formalized by defining a set of "instances of modelling structures" and operations which transform one instance into another. A modelling structure consists of a subset of the set of instances and a subset of the set of operations. In this way both the static and dynamic aspects of a modelling structure are described. In the description of this formalism we distinguish the meta-language for defining a modelling structure and the language for expressing the operations on instances of modelling structures. For example, saying that there exists a relation σ with some properties defined for a given modelling structure is part of the meta-language; while, saying that elements a and b in a particular instance of a modelling structure are σ related ($a \sigma b$) is part of the language for expressing operations on instances of modelling structures. (Appendix A to this paper describes the notational conventions used.)

The first section of the paper defines a set of "instances of modelling structures" and describes a graphical representation for elements in the set. The set of transformational operations is also discussed. The next section formally defines a modelling structure. The third section describes a number of properties which can be used for categorizing modelling structures. This includes a description of a set of primitive operations on instances of modelling structures (formal definitions for the operators are given in appendix B). To show the adequacy of this formalism, section four describes using the formalism a number of known structures, such as lists, trees, graphs, and arrays. This is followed by a discussion of possible uses for this formalism and a section discussing how this formalism relates to other work in the area of data structures.

Instances of Modelling Structures

An "instance of a modelling structure" is an ordered pair <n,v>, where n is the name and v is the value. A value is either a primitive indivisible object or a structured object. A structured object, or value, is a collection of objects, together with any structural relations between the objects. Each object is an instance of a modelling structure and thus has a name and value. Since this is a recursive definition (an object is a pair <n,v> where v is a collection of objects), we need some given set of indivisible values. These primitive indivisible values are instances of the particular data types provided as primitive by the programming language in which the modelling structures formalism is realized, for example, integers, reals, booleans and strings.

In the descriptions which follow the term "object" is used to mean an instance of a modelling structure. "Element" is used to mean an instance of a modelling structure which is a member of the value of another instance of a modelling structure. An object then is any arbitrary instance, while an element is an object treated as a member of a structure. "Structure" is used interchangably to mean an instance of a modelling structure or an arbitrary modellng structure. It should be clear from the context of its use which meaning applies. Finally, the term "value" is used to mean the set of elements of a particular structure.

Many instances of modelling structures may have the same value, but no two instances have the same name. The purpose for the name is to insure the uniqueness of each distinct instance of a modelling structure.

Formally, we define a set of instances of modelling structures by

$$\mathcal{V} = \{ \langle n,v \rangle \mid n \in \eta, v \in \mathcal{V}_p \text{ or } v \in \mathcal{V}_s \}$$

where η is a denumerable set of names, $\mathcal{V}_p$ is a set of primitive values, and $\mathcal{V}_s$ is a set of structured values.

The set of primitive values, $\mathcal{V}_p$, is a set of values

$$\mathcal{V}_p = \{v \mid v \text{ is a primitive indivisible value}\}$$

The set of structured values is defined by

$$\mathcal{V}_s = \{\langle \mathcal{O}, \mathcal{R}, \mathcal{D}, \mathcal{A}, P, M \rangle\}$$

where:

$$\mathcal{O} \subseteq \mathcal{V}$$
$$\mathcal{R} = \{\sigma \mid \sigma \subseteq (\mathcal{O} \times \mathcal{O}) \times 2^{\mathcal{V}}\}$$
$$\mathcal{D} = \{D_{x,n}\}$$
$$\mathcal{A} = \{\langle n', A_{x,n} \rangle\}$$

$\mathcal{O}$ is a set of objects, i.e. instances of modelling structures, which are the elements of the structure. (For particular applications of this formalism, it may be desirable to add the restriction that a structure may not be a member of itself nor may it be a member of any of its elements.) To denote that $\mathcal{O}$ is the set of objects for the structured value <n,v>, we will write $\mathcal{O}_n$ or $\mathcal{O}_{\langle n,v \rangle}$. This is also done for other parts of the elements of $\mathcal{V}_s$. $\mathcal{R}$ is a set of relations defined over the elements of the structure. The relations are expressed as an ordered pair $\langle\langle x,y \rangle, \mathcal{S} \rangle$, where $x,y \in \mathcal{O}$ are the two elements in the structure which are related, and $\mathcal{A}$ is a set of attributes of the defined relation. (For particular applications of this formalism there may be limitations on which elements of $\mathcal{V}$ may be in the attribute sets of relations for a given instance of a structured value.) $\mathcal{D}$ is a set of predicates which indicate whether an arbitrary element in $\mathcal{O}$ is a distinguished element. A distinguished element is an element in $\mathcal{O}$ which can be accessed based on some structural property. $\mathcal{Q}$ is a set of external accesses which are bound to elements in the structure. Elements in $\mathcal{Q}$ are primitive values ($\langle n, A_{y,\rho_1(x)}\rangle$ is an access to element x in y). P is the order predicate (if any) defined on the elements of $\mathcal{O}$. Finally, M is a predicate indicating whether the value of elements in $\mathcal{O}$ may be replicated (i.e. are there multiple copies of a value). If values may not be replicated then

$$\langle n,v\rangle,\langle n',v'\rangle \in \mathcal{O} \quad n \neq n' \Rightarrow v \neq v'$$

M = true means multiple copies (replication of values) are allowed in a structure.

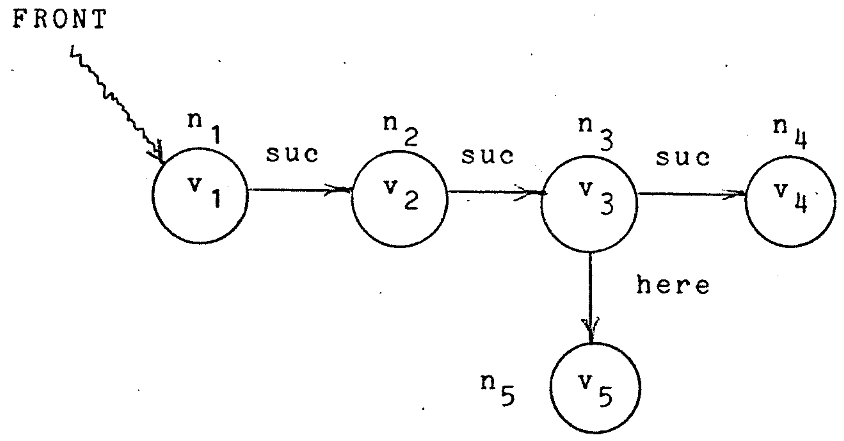
Relations between objects are part of the structure. They are the description of how the structure is put together. Some structures have elements which are unique in that particular relations are not defined for them. This may be a salient piece of information represented in the structure. Distinguished elements provide a mechanism for representing an invariant property of a modelling structure. In addition to this conceptual clarity, they also provide a focus for representing limitations on the operations performed on a given instance of a modelling structure. For example, consider a modelling structure with the one-to-one relation NEXT defined for all elements in a particular instance of the structure except x. This element x is special or distinguished in that it is the last element in the chain or list. Defining the distinguished element LAST to be the x such that NEXT(x) is undefined specifies the existence of this unique element. This approach makes clear those cases where a program accesses only this special element. Note that if a new object z were added to the modelling structure with NEXT(x) defined as z, then the LAST element would be z not x. Thus, a distinguished element is bound to the structure, not to a particular element. A distinguished element can be bound to one distinct element or no element. It may not refer to many elements. Furthermore, a distinguished element is not an object, so that it may not be manipulated as one. This means the predicate is not an object; however, the object returned by the predicate (i.e. the object referenced by the distinguished element at some point in time) is an object and may be manipulated as such.

Associated with each relation is a set of attributes. An attribute represents a piece of information associated with the relation between two specific elements in a structure. For example, to model a graph where the nodes correspond to cities and the edges correspond to roads between the cities, we might define a modelling structure in which the elements represent the nodes or cities and define a relation, named for instance ROAD, to represent the edges or roads. Associated with each road are a number of attributes describing the particular road, such as the distance between the cities, the speed limit for cars traveling on the road and the number of reststops between the cities. This data is associated with the road, and therefore the relation, and may be represented by attributes associated with the ROAD relation. The number of relations defined on a structure and the number of attributes for a particular value of a given relation is not limited.

Distinguished elements and external accesses are both mechanisms for referencing a particular element in a structured value. Whereas distinguished elements are bound to a structure, external accesses are bound to an element of a structure. The external access continues to reference the same element even if the structure is changed. It is bound to the element through the structure, so external accesses are restricted references. They may not be used to reference objects not in a structure, nor may an external access to element x in structure Y be used directly to reference x in structure Z. Note, there is no limit on the number of distinct structures which an object may be a member. An external access is an object (unlike distinguished elements) and can be used as such. This means they can be put into other structures.

An instance of a modelling structure is either ordered or unordered. If the structure is ordered then the predicate P defines the ordering. Within this formalism, ordering by predicate is the only explicit way to order the elements of a structure. However, it may be that an unordered structure is ordered implicitly by its use. (For example, the user may consider a structure with relation NEXT as used above to be ordered on that relationship.) In this case the user must manage the structure and its referencing if a special ordering on a sequence of references to elements of the structure is desired.

This definition of the set of instances of modelling structures captures the static nature of a modelling structure. An instance can be depicted graphically by a directed graph, where the nodes of the graph correspond to



$\langle \mathcal{O},\mathcal{R},\mathcal{D},\mathcal{A},P,M\rangle =$

$\langle\{\langle n_1,v_1\rangle,\langle n_2,v_2\rangle,\langle n_3,v_3\rangle,\langle n_4,v_4\rangle,\langle n_5,v_5\rangle\},$
$\{suc,here\},\{D_{FRONT}\},\{\},$ undefined, true$\rangle$

where:

$suc = \{\langle\langle\langle n_1,v_1\rangle,\langle n_2,v_2\rangle\rangle,\phi\rangle,\langle\langle\langle n_2,v_2\rangle,\langle n_3,v_3\rangle\rangle,\phi\rangle,$
$\langle\langle\langle n_3,v_3\rangle,\langle n_4,v_4\rangle\rangle,\phi\rangle\}$

$here = \{\langle\langle\langle n_3,v_3\rangle,\langle n_5,v_5\rangle\rangle,\phi\rangle\}$

$$D_{FRONT}(x) = \begin{cases} \text{false} & \text{if } \exists\sigma\exists y \ni \sigma(y)=x \\ \text{true} & \text{otherwise} \end{cases}$$

Figure 1: Example of Graphical Representation for an Instance of a Modelling Structure

the elements in the structure, the labeled edges ($\xrightarrow{\text{label}}$) are the named relations and the wavy lines ($\rightsquigarrow$) point to the distinguished elements. Figure 1 shows an example of a graphical representation for an instance of a modelling structure. This graphical representation is very similar to the one suggested by D´Imperio [1].

The set of operations on instances of modelling structures, named $\mathcal{F}$, is a set of partial functions which transform one instance into another. Examples of operators are: read the value of an element, insert an element, delete an element, and create an access to an element of a structure. Figure 2 is an example of deleting an element from a structure (in this case element n_3).

Modelling Structures

A modelling structure M is a subset of elements from $\mathcal{V}$ and a subset of elements from $\mathcal{F}$, the latter being the functions allowable on instances of M. Formally a modelling structure M is defined to be a pair $\langle V_M, F_M \rangle$, where

$$V_M \subseteq \mathcal{V}$$

$$F_M \subseteq \mathcal{F}\ ,\ f \in F_M \Rightarrow f: V_M \rightarrow V_M$$

This definition captures the static and dynamic nature of a modelling structure. Both the set of allowable instances

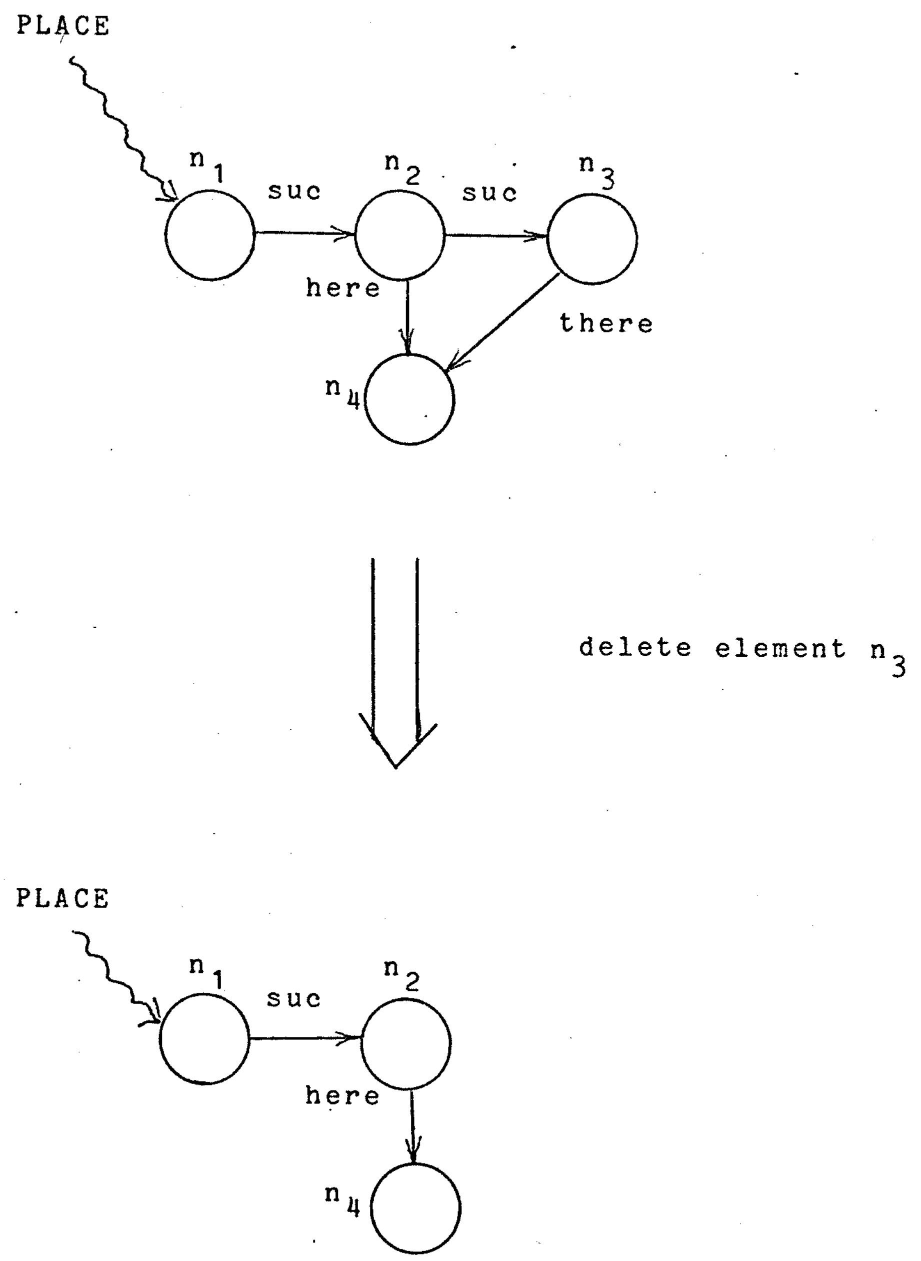


Figure 2: Example of Modelling Structure Transformation Deleting an Element

and the set of allowable operations (which must transform an allowable instance into another allowable instance) are necessary to characterize the modelling structure being used. For example, the distinction between a "stack" and a "one-way list" is in the operations by which objects are added to and deleted from the structure, in other words the set of allowable operations. An example in which the distinction is in the set of allowable instances is between a "one-way list" and a "two-way list."

One possible use of this formalism is as a guide in determining the presence of modelling structures in programs. The difficult problem with recognizing the use of an arbitrary modelling structure, in a program written in a given programming language, is that the characterization of $\langle V_M, F_M \rangle$ is divided between structure definition (state descriptions) and structure use (process descriptions). Furthermore, the description of V_M is not in general restricted to structure definitions, nor is the description of F_M restricted to structure uses. Also, similar modelling structures may be used in entirely different ways. These issues are discussed in more detail in the section on using the proposed formalism.

Properties of Modelling Structures

The previous sections of this paper have presented a formalization of our conception of modelling structures. While this formalization may be useful for analyzing or synthesizing modelling structures, it is incomplete as a framework for recognizing modelling structures used in a program, or for gathering the information necessary to choose implementation structures. In this section, eight properties of modelling structures useful for doing these tasks are discussed, with the emphasis on classification of modelling structures. The eight properties are:

1. Number of elements in a structure.
2. Type of elements in a structure.
3. Replication of element values in a structure.
4. Ordering of elements in a structure by their values.
5. Structural relations between elements.
6. Distinguished elements in a structure.
7. Referencing of elements in a structure.
8. Operations applied to a structure and its elements.

Because we are interested in recognizing whether the structures used in a program are examples of particular modelling structures, these properties are concerned with both the static and dynamic aspects of structures. Each of these properties is discussed below in more detail.

Statistics on the <u>number of elements</u> in a structure are not particularly important for classifying modelling structures. However, this information is extremely important for choosing efficient implementation structures.

The type of an object is not of direct concern in this paper, although it has been alluded to several times. We believe that in general the <u>type of elements</u> in a structure need not be a property of the modelling structure. (However, some programming languages impose restrictions on the types of objects which can be elements of a particular instance of a modelling structure as a result of implementation structure representation considerations.)

<u>Replication of element values</u> indicates whether two elements in a given structure may have the same value. A set is an example of a structure with no replication.

<u>Ordering of elements</u> in a structure presumes a predicate P. The ordering is specified by providing the predicate and is maintained "automatically" by the structure whenever an object is inserted or deleted.

The property of <u>structural relations</u> describes the relations defined for the structure. The properties of each relation defined for a structure and the relationship between the relations influences how the structure is referenced and changed. There are five distinct properties of a relation. (In this discussion it is sometimes convenient to think of the relation as a mapping from a domain to a range.) The first property of a relation determines whether an element can be related to one, or more than one, other element in the structure. A relation is one-one (1-1) if it and its inverse map an element of the structure to only one other element. A one-many (1-many) relation maps one element to many elements. A relation is many-one (many-1) if it maps many different elements to the same element. Note that the inverse of a 1-many relation must at least be many-1 and vice versa. Finally, a many-many relation is one which maps one element to many and many elements to one.

The second relation property establishes whether the relation for the domain and range is total, unique or partial. A total domain relation implies that all elements in the structure are in the domain of the relation. A unique domain relation implies that all elements in the structure except one are in the domain of the relation; and, a partial domain relation implies that zero or more elements in the structure may not be in the domain of the relation.

In a similar fashion the range of the relation is total, unique or partial. Thus, the possible choices for this property are one selection from each of the two lists (total domain, unique domain and partial domain, and total range, unique range and partial range).

The third property of a relation on a structure is a graph theoretic concept, namely, connected or not connected. A relation is connected (more precisely, a structure is connected with respect to a relation) if from each element all other elements in the structure can be reached by successive applications of the relation or its algebraic inverse (i.e. $\forall \langle x,y\rangle \in \sigma$, $\langle y,x\rangle \in \sigma^{-1}$). Another way to express this is to say that a relation is connected if the transitive closure of a relation and its inverse includes all elements in a structure. A relation is not connected if this condition does not hold.

The last two relation properties are algebraic concepts, namely, whether the relation is reflexive or symmetric. A relation σ is reflexive if for all elements x in the structure $\langle\langle x,x\rangle,\delta\rangle \in \sigma$. Thus, any time an element is added to the structure, $\langle\langle x,x\rangle,\phi\rangle$ is added in each reflexive relation. A relation σ is symmetric if for all elements x and y in the structure such that $\langle\langle x,y\rangle,\delta\rangle \in \sigma$, then $\langle\langle y,x\rangle,\delta\rangle \in \sigma$. Thus, whenever $\langle\langle x,y\rangle,\phi\rangle$ is added to a symmetric relation, $<<y,x>,\emptyset>$ is also added.

There are 288 different combinations of these five properties, of which 45 are realizable cases. The other property combinations result in contradictions and no examples can be created. Of the 45 realizable cases, 36 may be constructed and manipulated by the primitive structure operations defined below (16 connected and 20 not connected). The 16 realizable connected cases are listed in figure 3 and the 20 realizable not connected cases are listed in figure 4. 8 out of the 9 cases which cannot be constructed by the primitives are either unique domain-total range or total domain-unique range. There are no examples (having one or two nodes) of these cases which can be constructed in one <u>insert</u> or <u>relate</u> operation. The realizable cases can be grouped into three classes: list-like, tree-like and graph-like structures. The list-like structures are essentially lists and rings and are defined by 1-1 relations. The tree-like structures are essentially descendent and ancestor trees and are defined by 1-many and many-1 relations. The graph-like structures are essentially graphs and are defined by many-many relations. A complete listing of all distinct cases, examples of how they can be constructed by the primitive operators and PPL procedures implementing some of the primitive operators are

1-1

total domain	total range		
unique domain	unique range		

1-many

partial domain	total range		
partial domain	unique range		

many-1

total domain	partial range		
unique domain	partial range		

many-many

total domain	partial range		
unique domain	unique range		
unique domain	partial range		
partial domain	total range		
partial domain	unique range		
partial domain	partial range		
total domain	total range	symmetric	
partial domain	partial range	symmetric	
total domain	total range	reflexive	
total domain	total range	reflexive	symmetric

Figure 3: Realizable Connected Cases

1-1

total domain	total range
unique domain	unique range
partial domain	partial range

1-many

partial domain	total range
partial domain	unique range

many-1

total domain	partial range
unique domain	partial range
partial domain	partial range

many-many

total domain	total range		
total domain	partial range		
unique domain	unique range		
unique domain	partial range		
partial domain	total range		
partial domain	unique range		
partial domain	partial range		
total domain	total range	symmetric	
partial domain	partial range	symmetric	
total domain	total range	reflexive	
total domain	total range	reflexive	symmetric

Figure 4: Realizable Not Connected Cases

presented in a companion working paper [6].

The distinguished elements in the structure are described by predicates defined in terms of the relations defined for the structure.

An important property of a structure is how its elements are referenced. The formalism identifies four ways of referencing elements. These are: selection, distinguished element, external access and quantification.

There are two forms of selection referencing: element number and name selection. Element number referencing is used either to access elements in an ordered (in the sense of the ordering property) structure (get the i-th element) or an unordered structure (get the element named i). Element number referencing is a primitive concept in the formalism in the sense that the existance of the integers as element names may be assumed without explicit definition. In the unordered case the existence of a relationship between i and i+1 as integers and the elements referenced i and i+1 is not explicitly represented. Also, in the unordered case the element number may be a list of integers, thus introducing the notion of dimensionality. Element number referencing is denoted for unordered structures by

$$\langle n,v\rangle.i = \langle n',v'\rangle \in \mathcal{O}_n$$

and for ordered structures by

$$\langle n,v\rangle[i] = \langle n',v'\rangle \in \mathcal{O}_n$$

(In most programming languages both forms of element number referencing are depicted by the second form shown.)

The second form of selection referencing is name selection. Name selection referencing selects elements from a structure by their name. It is defined by

$$\langle n,v\rangle.n' = \langle n',v'\rangle \in \mathcal{O}_n$$

Distinguished element referencing references the element which satisfies the distinguished element predicate. Formally, a distinguished element is

$$x = \langle n',v'\rangle \in \mathcal{O}_n \quad D_{x,n}(\langle n',v'\rangle)=\text{true}$$

where $D_{x,n}$ is a member of $\mathcal{D}_n$.

Referencing by external access corresponds to binding an access function to an element in the structure. External accesses are formalized by a predicate which is true only for the element it is bound to,

$$x = \langle n',v'\rangle \in \mathcal{O}_n \quad A_{x,n}(\langle n',v'\rangle)=\text{true}$$

where $\langle n'',A_{x,n}\rangle$ is a member of $\mathcal{A}_n$. When the external access is evaluated it returns the element to which it is bound. External accesses are used for moving through structures such as lists, trees and graphs by following a sequence of related elements.

The last two mechanisms for referencing elements in a structure are content based, namely, universal and existential quantification. Universal quantification references all elements of a structure, possibly qualified by a predicate. For example,

$$\forall x \in Y \text{ such that } A$$

references all elements of the structure Y such that A(x) is true. Universal quantification might be included in a programming language as a looping control mechanism, e.g.

<u>forall</u> x <u>in</u> Y <u>suchthat</u> A <u>do</u>

might execute the range of the <u>do</u> repeatedly, with x bound in turn to each element of Y such that A(x) is true.

The second form of content based referencing is existential quantification. Existential quantification references an element (if it exists) which satisfies a predicate. For example,

$$\exists x \in Y \text{ such that } A$$

references an x in Y for which A(x) is true. This form of quantification might be included in a programming language as a mechanism for searching for an element in a structure, e.g.

<u>exists</u> x <u>in</u> Y <u>suchthat</u> A

binds to x an element of Y such that A(x) is true. Repeated execution of this statement does not generate all elements which qualify. It only finds an x which qualifies if it exists.

The predicate A used in both forms of quantification, as shown in the preceding examples, is defined only in terms of the values of the elements in the structure.

Any structure may use quantification referencing. Universal quantification may be thought of as implemented by coroutines generating the elements of the structure. If the structure is ordered the elements are generated in order, otherwise their order is not predictable.

The last property of structures is how they may be transformed, i.e. what <u>operations may be performed on a structure and its elements</u>. There are eleven primitive operations: <u>read</u>, <u>insert</u>, <u>delete</u>, <u>replace</u>, <u>createaccess</u>, <u>relate</u>, <u>unrelate</u>, <u>related</u>, <u>readattr</u>, <u>storeattr</u> and <u>assign</u>. The result of applying four of these operations (<u>insert</u>, <u>delete</u>, <u>relate</u> and <u>unrelate</u>) depends on the properties of

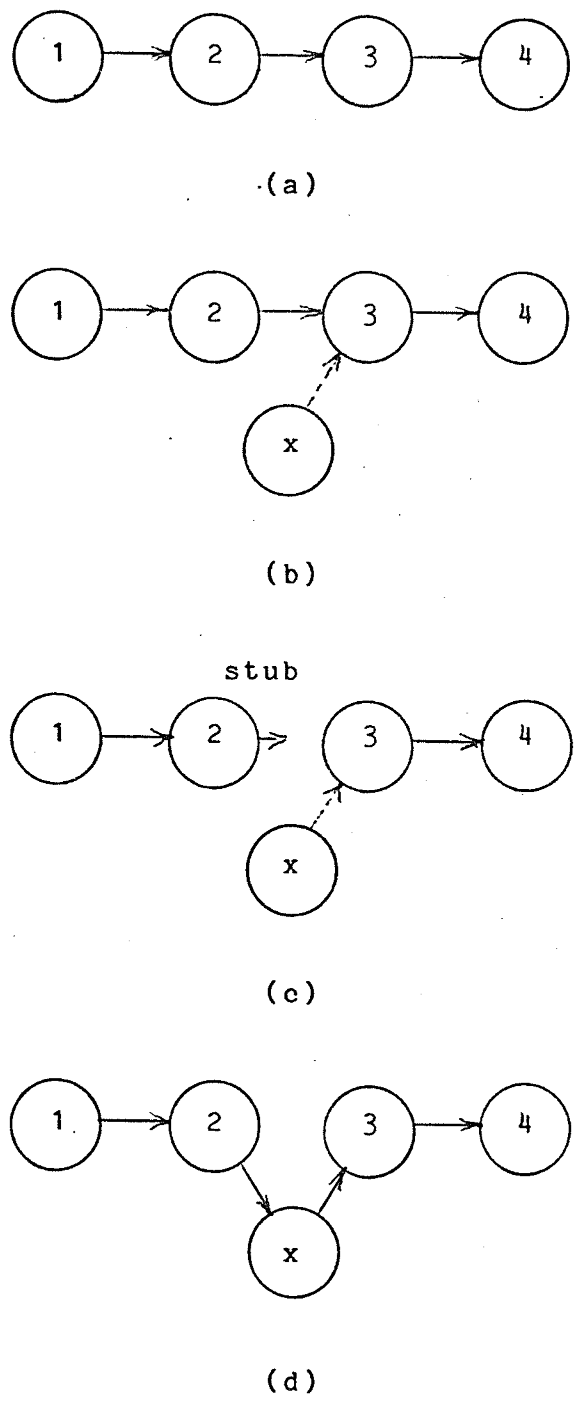


Figure 5: Sample Structure and _relate_ Operation

the relations defined on the structure. For example, if a non-empty structure had a 1-1, total domain, total range and connected relation defined on its elements, it would not be possible to <u>insert</u> a new object into the structure because how the new element is to be related to the other elements is ambiguous. However, in those cases where it is unambiguous, the operations transform the resultant structure into a legal structure. This can best be explained by an example. Suppose there is a 1-1, unique domain, unique range and connected relation defined on the elements of a structure, as shown in figure 5(a). A request to relate x to 3 would result in the illegal structure shown in 5(b). The structure is illegal because 2 related to 3 and x related to 3 implies the relation is not 1-1. Since the operation requires that x be related to 3, then 2 related to 3 must be removed as shown in 5(c). The structure shown in 5(c) is also illegal because it is not connected, nor is it unique domain and unique range. The rule for reconnecting the structure is to use any stubs, as shown in 5(c), to reconnect. Since there were only two possible reconnections, 2 to x and 4 to 1, and a stub 2 to something, the unambiguous transformation is to reconnect 2 to x. The resulting structure, after x has been inserted and related, is shown in figure 5(d). The transformation rule can be stated as:

> Add or delete relations described in the operation. If the resulting structure is legal then the operation is complete; otherwise, delete and/or add relations as necessary to restore a legal structure. If at any point an ambiguity arises (i.e. there is more than one way to delete or add relations as directed by stubs) the complete operation is undefined.

A rigorous definition for each of the operations is given in appendix B. For some cases when a relation is not connected there is an uncertainty as to the meanings of _delete_ and _relate_. As a result the definitions of these operators is meaningful only for connected relations.

The first operation is _read_. It returns the value of the referenced object.

The insertion operation, _insert_, inserts an object into a structure. If the structure is ordered the object is inserted so that the ordering is maintained.

The _delete_ operation removes an element from a structure and modifies all relations involving it.

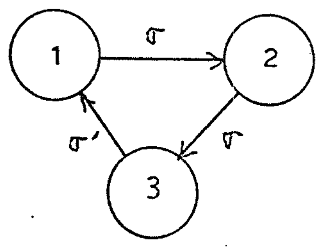
The _replace_ operation removes one element from a structure and replaces it with another. The definition of replace depends on whether the structure is ordered. If the structure is unordered the old element is removed (including external accesses) and the new element is inserted. All the relations defined for the old element are changed to relate to the new element. The only difference when the structure is ordered is that the ordering of elements is redefined in accordance with the order predicate rather than just by substitution.

The createaccess operation creates an external access to an element in a structure.

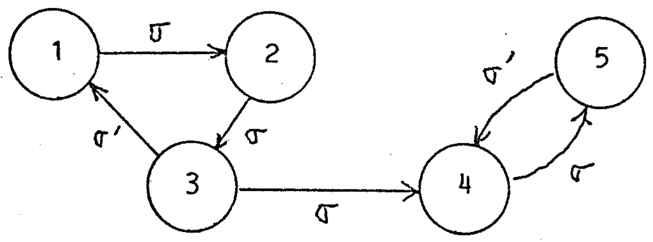
The relate operation relates elements. If either of the elements being related is not a member of the structure, it is also inserted. relate takes two arguments: the structure in which elements are to be related and an ordered set of triples $\langle x, \sigma, y \rangle$. For each of the triples in the ordered set in turn, x is related to y. If at any point during the processing of the triples an ambiguous transformation is encountered, the structure is restored to its original state, i.e. the complete operation is undefined. It should be remarked that in some circumstances the ordering of the triples may determine whether the operation is defined or not. After processing each triple the structure must be legal with respect to the relation named in the triple. Furthermore, at the completion of processing the triples, the structure must be legal with

σ 1-1, unique domain, unique range and connected

σ' many-many, partial domain, partial range and not connected



(a)



(b)

Figure 6: Example Structures and <u>relate</u> Operation

respect to all relations defined on the structure. An example may clarify these remarks. Figure 6(a) shows a structure with three elements and two relations: σ (1-1, unique domain, unique range and connected) and σ' (many-many, partial domain, partial range and not connected). The operation

$$\underline{\text{relate}}(\text{"structure"},\{\langle 5,\sigma',4\rangle,\langle 4,\sigma,5\rangle,\langle 3,\sigma,4\rangle\})$$

when applied to the structure in 6(a) is undefined because after processing the triple $\langle 4,\sigma,5\rangle$ the structure would be illegal (σ not connected). Note that after processing the first triple, $\langle 5,\sigma',4\rangle$, the structure is legal with respect to σ' but not with respect to σ. However, at that point the structure must be legal only with respect to the relation named in the triple, namely, σ'. After encountering the illegal transformation while processing the second triple, the structure is restored to its original state, as shown in 6(a), and the operator returns "undefined." By reordering the triples, the operation

$$\underline{\text{relate}}(\text{"structure"},\{\langle 3,\sigma,4\rangle,\langle 5,\sigma',4\rangle,\langle 4,\sigma,5\rangle\})$$

is defined and would transform the structure shown in 6(a) to the one shown in 6(b). Note that processing the first triple, $\langle 3,\sigma,4\rangle$, causes 4 to be inserted into the structure and processing the second triple causes 5 to be inserted. The operation

<u>relate</u>("structure",{<3,σ,4>,<5,σ',4>})

applied to the structure in 6(a) would be undefined, because at the end of processing all of the triples (both legal transformations) the resulting structure is illegal with respect to σ (with respect to σ 5 is not connected and σ is not unique domain or unique range).

<u>unrelate</u> is the inverse of <u>relate</u>. It removes the relation between two elements in a structure.

The <u>related</u> operation returns the object related to another by a particular relation.

<u>readattr</u> returns the attribute associated with a given relation.

The <u>storeattr</u> operation stores an attribute associated with a given relation.

The <u>assign</u> operation changes the value of a structure. Assign may not change a value of an element if that element is used in an order predicate for a structure. This insures that an ordered structure cannot be implicitly rearranged by an assignment operation.

Figure 7 lists the primitive operations and their arguments. These operations are not necessarily complete

```
read(object)

insert(object,structure)

delete(object,structure)

replace(object,object,structure)

createaccess(object,structure)

relate(structure,{<object,relation,object>,...})

unrelate(structure,{<object,relation,object>,...})

related(object,relation,structure)

readattr(structure,<element,relation,element>,attribute)

storeattr(structure,<element,relation,element>,
     <attribute,object>)

assign(object,object)
```

Figure 7: List of Primitive Operations

nor those that would appear in the syntax of a programming language. However, given an appropriate syntax and some other predicates (such as member of or related to), the meaning of most of the language constructs could be defined using these primitives.

Adequacy of the Formalism

This section demonstrates the adequacy of the proposed formalism by showing how a wide variety of modelling structures can be specified. These structures are specified in terms of the properties described in the previous section and the definition of the set of instances Υ. Obviously there are many different representations for each modelling structure. The purpose of this section is not to enumerate different representations, but to show how the formalism can cleanly represent different modelling structures.

In the examples that follow a "*" as an argument to a primitive operation means any reasonable value as an argument. Primitive operations listed without an argument list means any reasonable value for each argument.

Set

replication

    no.

ordering

    no.

relations

    none.

distinguished elements

    none.

referencing

    quantification

operations

    <u>read</u>

    <u>insert</u>

    <u>delete</u>

    <u>replace</u>

Ordered Set

replication

no.

ordering

yes.

relations

none.

distinguished elements

none.

referencing

quantification

element number

operations

<u>read</u>

<u>insert</u>

<u>delete</u>

<u>replace</u>

Sequence

replication

    yes.

ordering

    yes.

relations

    none.

distinguished elements

    none.

referencing

    quantification

    element number

operations

    <u>read</u>

    <u>insert</u>

    <u>delete</u>

    <u>replace</u>

Tuple

replication

yes.

ordering

no.

relations

none.

distinguished elements

none.

referencing

name selection

operations

<u>read</u>

<u>assign</u>

1-way List

replication

    yes.

ordering

    no.

relations

    suc (1-1, unique domain, unique range, connected)

distinguished elements

    "head" - suc^{-1}("head") = undefined

    "tail" - suc("tail") = undefined

referencing

    external access

    distinguished element

operations

    <u>read</u>

    <u>delete</u>

    <u>replace</u>

    <u>createaccess</u>

    <u>relate</u>

    <u>related</u>

2-way List

replication

yes.

ordering

no.

relations

suc (1-1, unique domain, unique range, connected)

pred = suc^{-1}

distinguished elements

"head" - suc^{-1}("head") = undefined

"tail" - suc("tail") = undefined

referencing

external access

distinguished element

operations

<u>read</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>related</u>

1-way Ring

replication

yes.

ordering

no.

relations

suc (1-1, total domain, total range, connected)

distinguished elements

none.

referencing

external access

operations

<u>read</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>related</u>

2-way Ring

replication

yes.

ordering

no.

relations

suc (1-1, total domain, total range, connected)

pred = suc^{-1}

distinguished elements

none.

referencing

external access

operations

<u>read</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>related</u>

Stack

replication

    yes.

ordering

    no.

relations

    next (1-1, unique domain, unique range, connected)

distinguished elements

    "head" - next("head") = undefined

referencing

    distinguished element

operations

    <u>read</u>

    <u>delete</u>

    <u>relate</u>(*,{<*,next,"head">})

Queue

replication

yes.

ordering

no.

relations

pred (1-1, unique domain, unique range, connected)

distinguished elements

"in" - pred("in") = undefined

"out" - pred^{-1}("out") = undefined

referencing

distinguished element

operations

<u>read</u>("out")

<u>delete</u>("out",*)

<u>relate</u>(*,{<"in",pred,*>})

Dequeue

replication

yes.

ordering

no.

relations

suc (1-1, unique domain, unique range, connected)

pred = suc^{-1}

distinguished elements

"end1" - suc^{-1}("end1") = undefined

"end2" - suc("end2") = undefined

referencing

distinguished element

operations

<u>read</u>("end1")

<u>read</u> ("end2")

<u>delete</u>("end1",*)

<u>delete</u>("end2",*)

<u>relate</u>(*,{<*,pred,"end1">})

<u>relate</u>(*,{<"end2",pred,*>})

Vector

replication

    yes.

ordering

    no.

relations

    none.

distinguished elements

    none.

referencing

    element number

operations

    <u>read</u>

    <u>assign</u>

Matrix

replication

    yes.

ordering

    no.

relations

    none.

distinguished elements

    none.

referencing

    element number (2-dimensional)

operations

    <u>read</u>

    <u>assign</u>

Array

replication

yes.

ordering

no.

relations

none.

distinguished elements

none.

referencing

element number (n-dimensional)

operations

<u>read</u>

<u>assign</u>

Tree

replication

yes.

ordering

no.

relations

desc (1-many, partial domain, unique range, connected)

ansc (many-1, unique domain, partial range, connected)

ansc = $desc^{-1}$

distinguished elements

"root" - ansc("root") = undefined

referencing

external access

distinguished element

operations

<u>read</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>related</u>

Binary Tree

replication

yes.

ordering

no.

relations

left (1-1, partial domain, partial range, not connected)

right (1-1, partial domain, partial range, not connected)

ansc (many-1, unique domain, partial range, connected)

ansc = $\text{left}^{-1} \cup \text{right}^{-1}$

distinguished elements

"root" - ansc("root") = undefined

referencing

external access

distinguished element

operations

<u>read</u>

<u>delete</u>

replace

createaccess

relate

related

Graph

replication

yes.

ordering

no.

relations

x_i (any properties)

$y_i = x_i^{-1}$

distinguished elements

may be defined.

referencing

external access

distinguished element

operations

<u>read</u>

<u>insert</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>unrelate</u>

related

readattr

storeattr

Digraph

replication

yes.

ordering

no.

relations

x_i - (any properties)

distinguished elements

may be defined.

referencing

external access

distinguished element

operations

<u>read</u>

<u>insert</u>

<u>delete</u>

<u>replace</u>

<u>createaccess</u>

<u>relate</u>

<u>unrelate</u>

<u>related</u>

readattr

storeattr

Uses for the Formalism

Defining a formalism is not very interesting unless it can be used, either to analyze or synthesize the objects with which it is concerned or to elucidate some of their common properties. Furthermore, the formalism itself should show an interaction of properties. A formalism composed of disjoint parts which does not reveal interesting relationships between the objects with which it deals is not itself very interesting. We feel that the formalism presented here meets this last criterion, particularly in the way relations, distinguished elements, referencing and ordering interrelate to define sets of similar modelling structures. One purpose for defining this formalism is to investigate a mechanism for recognizing whether the definition and use of a particular instance of a structure in a program is the use of some known modelling structure. By developing a catalog of descriptions for known structures and an algorithm for deducing from the program the properties used for describing structures in the formalism, it is possible to do this. Notice that the properties used in the preceding example specifications of modelling structures can in most cases be easily determined by a static analysis of a program (assuming a semantically well-behaved language syntax). Thus, the catalog (including alternative representations for the same structures) and algorithm could comprise the modelling structure recognition component of the system described in reference 5. Since a given modelling structure can be expressed in many different ways and the general problem of recognizing any arbitrary representation is unsolvable, one measure of the quality of the formalism is whether a reasonable number of cases can be recognized.

There is a problem with the modelling structures recognition algorithm. Because objects being related using the <u>relate</u> operation may or may not be members of the structure and because in general it is not possible to determine whether the object is or is not already a member, the ability to recognize those modelling structures where there is a limitation is uncertain (e.g. $\langle x,\sigma,y \rangle$, x must be and y may not be a member of the structure). There are three ways to deal with this problem. First, in those cases where it is uncertain, the system could ask the user. A second possibility would be to build a complicated pattern matching program to recognize some percentage of the uncertain cases. In those cases where the pattern matcher could not determine whether an object is or is not already in a structure, the recognition algorithm would proceed assuming that either could be true, implying that some known modelling structures may not be recognized. The third possibility would be to constrain the programming language syntax and semantics in a way that forces the programmer to specify whether an object is or is not already a member. Each of these alternatives has advantages and disadvantages and will not be discussed further in this paper.

The catalog of known modelling structures used by the recognition algorithm can serve as the store of predefined modelling structures an experienced programmer may use. This mechanism can also be used to check the consistency of structure use in a program by comparing the structure description deduced from the program with the catalog description and by checking the inner consistency of the deduced descriptions themselves. It should also be possible to recognize when the structure actually used is different from the one the programmer thinks he is using (e.g. if a supposed list is in fact a stack or queue).

Other areas where this formalism can be used are in synthesizing and analyzing particular modelling structures. It can also be useful as a tool for comparing the structure facilities provided in different programming languages. For example, what properties and operators are provided, what limitations are imposed and what concepts are combined? Another use for this formalism is as the framework and knowledge base for interactively synthesizing modelling structures.

Relationship to Previous Work

There has been a significant amount of work on data structures in the last few years. In this section we discuss the relationship between some of this previous work on formalizing and using data structures and our proposed modelling structures formalism. The work by Kapps [3], Mealy [4] and Turski [9] has been directed towards a theory of data structures emphasizing the meaning of data and the fundamentals of structures and computation. Many of the concepts represented in these abstract models are used in our formalism. Earley [2], Shneiderman and Scheuermann [7], and Taft and Standish [8] are representative of the extensive work on data definitional facilities and structure operations in programming languages. Because the proposed formalism will be the basis of a sample programming language for research on the generation of efficient implementation structures, it is important to consider how usable such a language will be. In this paper the emphasis is on the automatic recognition of structures used in a program. However, we feel that data definitional facilities based on our formalism may also prove more usable than some presently available facilities. Lastly, we discuss the work by D´Imperio [1] on problem-solving tools, particularly her modelling structures formalism.

Kapps defines a data structure as a pair $<\mathcal{C} \xrightarrow{f} \mathcal{D}, \mathcal{R}>$, where $\mathcal{C}$ is a set of storage cells, $\mathcal{D}$ is a set of data values and $\mathcal{R}$ is a set of relations on $\mathcal{C}$ (hence on $\mathcal{D}$). Computation is then a process mapping a program data space structure to another structure. He factors out equivalent data structures by defining an algebraic category of morphisms over the class of data structures.

This formalism is used to discuss programming systems (a subcategory of data structures which are directly representable in the system), solution of problems (mapping from the subcategory of input data structures to the subcategory of solutions to elements in the input subcategory) and mechanizing the generation of problem solution procedures (mapping the data values to $\mathcal{C}$, mapping the relations $\mathcal{R}$, and mapping the operations). Kapps acknowledges that the mapping of relations and operations is intimately related to and influences the execution efficiency of the computation. He can not carry this much further since he does not discuss in detail the mechanisms for referencing and operating on data structures. They are treated as abstract, presumably partial, functions. Our proposed formalism differs from Kapps' work on two points. The first is that he presumes a physical storage medium (in his notation $\mathcal{C}$) in which data values are stored, and second, concepts which are represented explicitly in our formalism are subsumed in unspecified functions and relations.

Mealy discusses a theoretical model for data and data processing. The model is based on entities (objects in the real world), values (values of attributes of entities), data maps (mappings of values to attributes of entities) and procedural maps (operations on data maps which change the data maps). Structural data maps are special data maps whose value set is the set of entities (in essence a structural data map specifies which entities are elements of the structure). The model also defines access functions (procedural mechanisms for accessing entities), data organization (how data is mapped into a physical storage medium) and data description (specification of data representation).

Mealy uses this model to discuss problems in three areas: representation independence, language extension and variable binding time. Briefly, he argues that programmers should have more explicit control of data representation, that languages for extending data types are better than languages with expanded numbers of data types and lastly, that languages and systems should make more use of stored, explicit data descriptions.

Mealy's formalism is different than Kapps' in that Mealy does not require a structure to be represented in some storage medium. He represents relations by data maps and does not explicitly associate them with a specific structure. He also introduces the notion of access functions; although, he does not make a distinction between accesses based on structural properties (our distinguished elements) and accesses to specific elements (our external accesses). Furthermore, his access functions are features of individual procedures. Similarly to Kapps, he subsumes some of the aspects explicitly represented in our formalism (replication, ordering, referencing and operations) in unspecified mappings.

Turski's model represents what are termed "unambiguous structures" composed of objects each with a distinct name. This name can be used as references, structural links (i.e. as objects themselves) or preimages of storage allocation mappings. The model, similar to Mealy, does not presume storage of the data values. The mapping of a data structure to an addressable storage meduim (via traditional hardware addressing mechanisms, namely, indexed, relative and indirect) is described for three classes of structures:

array-like, key-ordered list-like and threaded list-like. Our formalism, like Turski's, uses named objects as its primitive elements; however, names are restricted to being used as references.

Most workers in data structures agree that there are various levels at which structures and operations on the structures can be represented. The choice of at what level of generality the data definitional facilities are realized in a programming language has a major impact on many aspects of the language and its use, including but not restricted to: the ease of expressing programs, the possibilities for proving programs correct, the transportablitiy of programs, program execution efficiency, and the reliability and maintainability of programs. Obviously, machine and assembly languages provide the greatest freedom, and require the greatest effort on the part of the programmer, in describing and using data structures. High level languages (for example FORTRAN, ALGOL-60, COBOL, SNOBOL, COMIT, SLIP and BASIC) provided limited facilities, limiting either the types of representable structures and/or the available operations. This reduced the effort required by the programmer, but also limited the scope of problems which could easily be programmed. Unfortunately, this resulted in a plethora of new languages and extensions to old languages for different problem domains.

As a result research in programming languages turned to the design of languages and systems with data structuring facilities which would attract a wider audience of users. One effort in this direction was PL/1. It provided facilities for defining complicated structures, but the use of these structures was cumbersome because neither the structures, nor their associated operators, were really a part of the language as had been the case with the special purpose high level languages. This lead to the design of the extensible languages (for example ALGOL-68, PPL, ECL and VERS). It should be pointed out that the virtues of the extensible languages, the ability to construct new data types and embed new operators in the language, had appeared much earlier in LISP and to some extent in the IPL series. This was accomplished by making operators, procedures and functions syntactically appear the same. Thus whether a given operator was primitive in the language or had been added by the user was transparent. The problem now is at what level of generality the primitives for constructing and using structures should be defined.

PPL, as described in the paper by Taft and Standish, uses the currently popular data defintional facilites: rowing (forming sequences of elements of the same type using element number referencing), structuring (forming finite sized structures of possible different types using name selection referencing), uniting (forming a new data type which is the union of several other types -- when the structure is created or assigned to, its type is fixed) and referencing (forming classes of pointers to elements of other classes). These mechanisms provide a concise, easy to understand set of data definitional facilities. Unfortunately, they place a considerable burden on the programmer when he wants to use certain types of structures, such as: dynamic collections of dissimilar objects, ordered structures (the programmer must maintain the structure), structures with no replication, and multiple relations on elements in a structure. In other cases it requires a well documented, easy to access library of commonly used structures (such as stacks, trees and lists) or the programmer must reprogram these structures each time they are used.

Shneidernam and Scheuermann discuss a higher level set of data definitional facilites. They suggest a facility which allows arbitrary compositions of structures, with essentially two primitive types of structures: trees and lists. They also provide higher level primitive operators for most of the meaningful operations: insert, delete, copy, replace, interchange and search. While these facilities will probably be very useful to those problem domains with structures similar to the ones defined, the number of users will most likely be limited.

Earley in his VERS system provides a rich data definitional facility and a powerful set of iterative operators. The data structuring mechanisms provided include: tuples (corresponds to structured types), sets (homogeneous, unordered collections of elements), relations (sets of tuples), functions (binary relations) and sequences (ordered collections of homogeneous elements with replication). Sequences are used to model vectors, strings and lists. With each structure type there are many useful primitive operators defined, e.g. set union and intersection, a next and prev operator for sequences and insert and delete operators for both sets and sequences. It is difficult to tell which of the two facilities, Earley´s or ours, would be most usable over a wide class of problems. However, it does appear that using Earley´s formalism it may not be as easy to automatically recognize the use of known structures.

The proposed formalism is closest to the work of D´Imperio. Her formalism is based on a set of primitive nodes (in our terminology Υ_p) and structures of nodes which vary in three ways: internal ordering of nodes within a structure, external accesses to elements in a structure and references between structures and units. The basic unit is a list, described by a structure´s internal ordering property. External access corresponds to our notion of distinguished elements (presumably they are explicity updated by the primitive operations used). References are essentially access paths between elements of different structures. Her formalism is a tool for human problem-solving, and to our knowledge has not been used as a basis for a programming language data definitional facility. We have extended her formalism in several significant ways: by making the property of internal ordering optional (her formalism did not require that a structure be ordered but implicitly it always was by the way the structure was represented), by allowing multiple relations to be defined for elements in one structure, by binding external accesses to structures, by allowing one object to be in many structures at once, by allowing structures with no replication of elements to be defined, by introducing hierarchy in a different way (allowing structures to be treated as nodes in other structures) and by allowing the definition of arbitrary relations between objects (corresponds in a limited way to her references). We have also rigorously defined a concise set of primitive operators.

Summary

A formalism for describing the properties of modelling structures is defined, including detailed descriptions of the static (properties) and dynamic (operations) aspects of a rich class of structures. The adequacy of the formalism is demonstrated by the ability to represent a wide variety of known modelling structures. Some possible uses for the formalism are proposed and the relationship between this formalism and selected previous work on data structures discussed.

References

1 D´Imperio, M. E. Information Structures: Tools in Problem-Solving. unpublished paper (July 1969).

2 Earley, J. Relational Level Data Structures for Programming Languages. Computer Science, U. C. Berkeley (1973).

3 Kapps, C. A. SPRINT A Programming Language with General Structure. PhD Thesis Moore School Report No. 71-18, The Moore School of Electrical Engineering, Univ. of Penn. (August 1970), p 94-123.

4 Mealy, G. M. Another Look at Data. Proc. AFIPS 1967 FJCC, 31 (May 1967), p 525-534.

5 Rowe, L. A. A Formalization of Modelling Structures and the Generation of Efficient Implementation Structures. Dissertation Proposal, Department of Information and Computer Science, U. C. Irvine (May 1974).

6 Rowe, L. A. Modelling Structures Formalism -- Structural Relation Case Examples. unpublished working paper, Department of Information and Computer Science, U. C. Irvine (December 1974).

7 Shneiderman, B. and P. Scheuermann. Structured Data Structures. Comm. ACM 17, 10 (October 1974), p 566-574.

8 Taft, E. A. and T. A. Standish. PPL User's Manual. Aiken Lab, Harvard University (January 1971).

9 Turski, W. M. A Model for Data Structures and its Applications. ACTA Informatica 1 (1971),p 26-34.

Appendix A: Notational Conventions

The conventions used in this paper are that capital script letters represent sets $(\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots)$, capital block letters are predicates (A, B, C,...), small letters are functions and variables (a, b, c,...) and that the greek letter sigma represents a relation (σ).

A relation σ is a subset of the cartesian product of 2 sets. It can be represented by a set of ordered 2-tuples,

$$\sigma = \{\langle x,y\rangle \mid x \text{ is } \sigma \text{ related to } y\}$$

That an element $\langle x,y\rangle$ is a member of a particular relation is denoted by $x\sigma y$ or $\sigma(x) = y$.

A function or relation f mapping domain $\mathcal{A}$ to range $\mathcal{B}$ is shown by

$$f:\mathcal{A} \rightarrow \mathcal{B}$$

Two projection functions are defined, namely, ρ_1 and ρ_2. ρ_i projects the i-th component of the ordered tuple. thus,

$$\rho_2(\langle x,y\rangle)=y$$

In the operator formalism, set union $(\cup)$, subtraction $(\backslash)$ and a procedural assignment $(\leftarrow)$ are used, e.g.

$$\mathcal{A} \leftarrow \mathcal{A} \cup x$$

makes $\mathcal{A}$ the set union of $\mathcal{A}$ and x. ϕ denotes the null set and $2^{\mathcal{A}}$ denotes the power set of $\mathcal{A}$ (the set of all possible subsets of $\mathcal{A}$).

Appendix B: Operator Definitions

In this appendix formal definitions for the eleven primitive operators are presented. The operators are defined by procedures which transform a legal structure into a legal structure. If the structure to be operated on is illegal (e.g. a total domain relation for a structure is not total), then the result of applying an operator is indeterminate. All operators, except <u>read</u>, <u>related</u> and <u>readattr</u>, return "defined" or "undefined" depending on whether the operation is performed or not performed. In those cases where applying an operator returns "undefined" and the structure was transformed, the undoing of the operation is not shown.

The definitions rely heavily on the notational conventions established in appendix A. Keywords and primitive functions are underlined. The meanings for most of the primitive functions are described in the comments associated with the definitions. There are two primitive boolean functions, <u>def</u> and <u>undef</u>, used in the definition of <u>relate</u> that may need more explanation. The functions define or undefine a relation between two elements of a structure. However, if the relation to be defined (undefined) has been undefined (defined) by the processing of a previous triple, the function returns <u>false</u> which results in the operator returning "undefined."

The arguments to the operators are those shown in the list of primitive operations (figure 7).

READ

```
read(x)                               read the value of x
   return $\rho_2(x)$                 return value part of argument
end
```

INSERT

```
insert(x,y)                                                   insert x into y
  if $y \in \Upsilon_p$ then return "undefined"               may not insert into primitive structure
  if not($M_y$) or $\nexists z \in \mathcal{O}_y \ni \rho_2(z)=\rho_2(x)$ then
                                                              if structure allows repetition or x
                                                              is not already in y then put x into y
    if $(\mathcal{O}_y \neq \phi)$ and $(\exists \sigma \in \mathcal{R}_y \ni (\sigma$ connected or
                                                              if structure not empty and there exists
      $\sigma$ unique range or $\sigma$ unique domain))       a connected or unique domain or range
      then return "undefined"                                 relation on y, then op not defined
    $\forall \sigma \in \mathcal{R}_y$ do                     for all relations on y
      if $\sigma$ reflexive or $\sigma$ total domain          if there is a reflexive or total domain
         or $\sigma$ total range then                         or range relation on y then add x
        $\sigma \leftarrow \sigma \cup \langle\langle x,x\rangle,\phi\rangle$
                                                              related to x
      if $\mathcal{O}_y=\phi$ then                            if structure is empty and
        if $\sigma$ unique domain then $t_\sigma \leftarrow x$
                                                              unique domain, then tail ($t_\sigma$) is x
        if $\sigma$ unique range then $h_\sigma \leftarrow x$
                                                              unique range, then head ($h_\sigma$) is x
    if $P_y$ defined then order(x,y)                          if y is ordered structure then establish
                                                              ordering of x in structure
    $\mathcal{O}_y \leftarrow \mathcal{O}_y \cup x$           put x in object set of y
    return "defined"                              operation is defined
  end
```

DELETE

```
delete(x,y)                                              delete x from y

  if $y \in \Upsilon_p$ or $x \notin \Theta_y$ then return "undefined"    if y is a primitive structure or x is
                                                         not an element of y then op is not
                                                         defined

  $\forall \sigma \in \mathcal{R}_y$ do                                   for all relations on y do

    case of

      $\sigma$ 1-1:                                      $\sigma$ 1 to 1 relation

        if $\exists \langle\langle v,x\rangle,\mathcal{S}\rangle \in \sigma$ then           if there exists a v related to x

          $\sigma \leftarrow \sigma \setminus \langle\langle v,x\rangle,\mathcal{S}\rangle$                   undefine it

          if $\exists \langle\langle x,u\rangle,\mathcal{S}'\rangle \in \sigma$                 if there exists x related to u

            then $\sigma \leftarrow (\sigma \setminus \langle\langle x,u\rangle,\mathcal{S}'\rangle) \cup \langle\langle v,u\rangle,\phi\rangle$     undefine it and define v related to u

            else if $\sigma$ unique domain then $t_\sigma \leftarrow v$      otherwise v is tail

          exitcase                                       leave case statement

        if $\exists \langle\langle x,u\rangle,\mathcal{S}\rangle \in \sigma$ then           no y related to x, if there exists x
                                                         related to u then

          $\sigma \leftarrow \sigma \setminus \langle\langle x,u\rangle,\mathcal{S}\rangle$                     undefine it

          if $\sigma$ unique range then $h_\sigma \leftarrow u$            and if appropriate, u is head

        exitcase                                         leave case statement
```

```
$\sigma$ 1-many:                                          $\sigma$ relation is 1 to many
  if $\sigma$ unique range and $h_\sigma = x$ then        if   unique range and x is head then
    if $|\{v \mid \langle\langle x,v\rangle,\mathcal{S}\rangle \in \sigma \wedge v \neq x\}| > 1$    only 1 element may be in relation to x
      then return "undefined"                             otherwise operation is not defined
      else
        if $\exists\langle\langle x,v\rangle,\mathcal{S}\rangle \in \sigma$ then    if defined and v exists
          $\sigma \leftarrow \sigma \setminus \langle\langle x,v\rangle,\mathcal{S}\rangle$    undefine relation and
          $h_\sigma \leftarrow v$                         make v new head
        exitcase                                          leave case statement
  $\langle\langle v,x\rangle,\mathcal{S}\rangle \in \sigma$    let v be the element in relation to x
  if $x = v$ then                                         if x is equal to v and there is more
    if $|\{u \mid \langle\langle x,u\rangle,\mathcal{S}'\rangle \in \sigma \wedge u \neq x\}| > 1$    than one element related to v then
      then return "undefined"                             operation is not defined
      else
        if $\exists\langle\langle x,u\rangle,\mathcal{S}'\rangle \in \sigma$ then    if u exists,
          $\sigma \leftarrow (\sigma \setminus \langle\langle x,u\rangle,\mathcal{S}'\rangle) \cup \langle\langle u,u\rangle,\phi\rangle$    undefine x related to u and define u
                                                          related to u
    exitcase                                              leave case statement
  $\forall\langle\langle x,u\rangle,\mathcal{S}'\rangle \in \sigma$ do    for all elements related to x, unrelate
```

```
    $\sigma \leftarrow (\sigma \setminus \langle\langle x,u\rangle,\mathcal{S}'\rangle) \cup \langle\langle v,u\rangle,\phi\rangle$      and relate v to elements
  $\sigma \leftarrow \sigma \setminus \langle\langle v,x\rangle,\mathcal{S}\rangle$                                   undefine v related to x
  exitcase                                                  leave case statement
$\sigma$ many-1:                                                 $\sigma$ relation is many to 1
  if $\sigma$ unique domain and $t_\sigma = x$ then                        if unique domain and x is tail then only
    if $|\{v \mid \langle\langle v,x\rangle,\mathcal{S}\rangle \in \sigma \wedge v \neq x\}| > 1$                       1 element may be in relation to x
      then return "undefined"                               otherwise operation is not defined
      else
        if $\exists \langle\langle v,x\rangle,\mathcal{S}\rangle \in \sigma$ then                          if there is only one element in relation
          $\sigma \leftarrow \sigma \setminus \langle\langle v,x\rangle,\mathcal{S}\rangle$                             to x, undefine v related to x and make
          $t_\sigma \leftarrow v$                                          v the new tail
        exitcase                                            leave case statement
  $\langle\langle x,v\rangle,\mathcal{S}\rangle \in \sigma$                                            let v be the element related to x
  if x=v then                                               if v equals x and there is more than
    if $|\{u \mid \langle\langle u,x\rangle,\mathcal{S}'\rangle \in \sigma \wedge u \neq x\}| > 1$                      1 element related to u then
      then return "undefined"                               the operation is not defined
      else
```

```
            if $\exists\langle\langle u,x\rangle,\mathcal{S}'\rangle\in\sigma$ then           otherwise, if only 1 u exists then
              $\sigma\leftarrow(\sigma\setminus\langle\langle u,x\rangle,\mathcal{S}'\rangle)\cup\langle\langle u,u\rangle,\phi\rangle$         undefine u related to x and define u
                                                       related to u
        exitcase                                       leave case statement
      $\forall\langle\langle u,x\rangle,\mathcal{S}'\rangle\in\sigma$ do                           for all elements in relation to x
        $\sigma\leftarrow(\sigma\setminus\langle\langle u,x\rangle,\mathcal{S}'\rangle)\cup\langle\langle u,v\rangle,\phi\rangle$               undefine relation to x and relate to v
      $\sigma\leftarrow\sigma\setminus\langle\langle x,v\rangle,\mathcal{S}\rangle$                            undefine x related to v
      exitcase                                         leave case statement
    $\sigma$ many-many:                                     $\sigma$ relation is many to many.
      if $\sigma$ reflexive then                           if reflexive, undefine x related to x
        $\sigma\leftarrow\sigma\setminus\langle\langle x,x\rangle,\mathcal{S}\rangle$
      $\forall\langle\langle x,v\rangle,\mathcal{S}\rangle\in\sigma$ do                           for all v in relation to x
        $\sigma\leftarrow\sigma\setminus\langle\langle x,v\rangle,\mathcal{S}\rangle$                          undefine x related to v
        if $x\neq v$ then                                  if x is not equal to v
          if $\sigma$ total range and                      if total range and there does not exist
            $\nexists\langle\langle u,v\rangle,\mathcal{S}'\rangle\in\sigma$ then                u related to v then if structure has
            if $|\sigma_y|>2$                                  more than 2 elements operation is not
              then return "undefined"                  defined, otherwise define v related to v
              else $\sigma\leftarrow\sigma\cup\langle\langle v,v\rangle,\phi\rangle$
    if $\sigma$ unique range and                 if unique range and there does not exist
      $\nexists\langle\langle u,v\rangle,\delta'\rangle\in\sigma$ then          another element for which v is in range
     if $h_\sigma = x$                            then if x is the head, make v the head
       then $h_\sigma \leftarrow v$
       else return "undefined"                   otherwise more than 1 head
 if $\sigma$ unique range and $h_\sigma = x$ and     if unique range and no new head then op
    $|\theta_y| > 1$ then                          is not defined
   return "undefined"
 $\forall\langle\langle v,x\rangle,\delta\rangle\in\sigma$ do                 for all elements that x is in the range
   $\sigma \leftarrow \sigma \setminus \langle\langle v,x\rangle,\delta\rangle$                 undefine relation with x
   if $\sigma$ total domain and                 if total domain and there does not
      $\nexists\langle\langle v,u\rangle,\delta'\rangle\in\sigma$ then          exist element for which v is in domain
     if $|\theta_y| > 2$                           then operation is not defined, otherwise
       then return "undefined"                   if only 2 or less elements define v to v
       else $\sigma \leftarrow \sigma \cup \langle\langle v,v\rangle,\phi\rangle$
   if $\sigma$ unique domain and                if unique domain and there does not
      $\nexists\langle\langle v,u\rangle,\delta'\rangle\in\sigma$ then          element in range of v then if x was head
     if $t_\sigma = x$                            make v head, otherwise operation is not
```

```
            then $t_\sigma \leftarrow v$                              defined
            else return "undefined"
      if $\sigma$ unique domain and $t_\sigma = x$ and                if unique domain and x was head, if no
         $|\mathcal{O}_y| > 1$ then return "undefined"               new head then operation not defined
      if $\sigma$ connected and $|\mathcal{O}_y| > 2$ then            if relation connected and more than 2
        if notconn(y) then                                 elements in structure, must still be
          return "undefined"                               connected, otherwise op not defined
      exitcase                                             leave case statement
    endcase                                                end of case statement
  $\forall \langle n,A\rangle \in \mathcal{A}_y \ni A(x)=$true do                   remove external accesses to x
    $\mathcal{A}_y \leftarrow \mathcal{A}_y \setminus \langle n,A\rangle$
  if $P_y$ defined then unorder(x,y)                       if structure ordered remove ordering of
                                                           x
  $\mathcal{O}_y \leftarrow \mathcal{O}_y \setminus x$                                     remove x from object set of structure
  return "defined"                                         operation is defined
end
```

REPLACE

```
replace(x,y,z)                                                    replace x with y in structure z
  if $z \in \Upsilon_p$ or $x \notin \mathcal{O}_z$ or $y \in \mathcal{O}_z$ or                      if structure is primitive or x is not in
      (not($M_z$ and $\exists u \in \mathcal{O}_z \ni \rho_2(u)=\rho_2(y)$)) then        the structure or y is already in z
    return "undefined"                                            or structure does not allow replication
                                                                  and value of y already in z then op is
                                                                  not defined
  $\forall \tau \in \mathcal{R}_z$ do                                                   for all relations on the structure
    $\forall \langle\langle u,x\rangle,\mathcal{S}\rangle \in \tau$ do $\tau \leftarrow (\tau \setminus \langle\langle u,x\rangle,\mathcal{S}\rangle) \cup \langle\langle u,y\rangle,\mathcal{S}\rangle$    redefine relations involving x to
    $\forall \langle\langle x,u\rangle,\mathcal{S}\rangle \in \tau$ do $\tau \leftarrow (\tau \setminus \langle\langle x,u\rangle,\mathcal{S}\rangle) \cup \langle\langle y,u\rangle,\mathcal{S}\rangle$    relate to y
  if $P_z$ defined then                                           if structure is ordered remove ordering
    unorder(x,z)                                                  for x and establish ordering for y
    order(y,z)
  $\forall \langle n,A\rangle \in \mathcal{A}_z \ni A(x)=$true do                               redefine external accesses to x to
    $\mathcal{A}_z \leftarrow \mathcal{A}_z \setminus \langle n,A\rangle$                                       access y (use call on primitive
    createaccess(y,z)                                             operation createaccess)
  $\mathcal{O}_z \leftarrow (\mathcal{O}_z \setminus x) \cup y$                                          remove x from object set and add y
  return "defined"                                                operation is defined
```

```
end
```

CREATEACCESS

```
createaccess(x,y)                                 create an external access to x in y
  if $y \in \mathcal{V}_p$ or $x \notin \mathcal{O}_y$ then                     if y is a primitive structure or x is
    return "undefined"                            not an element of y then the operation
                                                  is not defined
  createfunction A(x)=true                        create the access function
  $\mathcal{A}_y \leftarrow \mathcal{A}_y \cup$ <gen,A>                        add the access to the set of external
                                                  accesses for y
  return "defined"                                operation is defined
end
```

RELATE

```
relate(x,y)                                          in structure x relate <obj,rel,obj>
                                                     triples in y

  if $x \in \mathcal{V}_p$ then                              x must be a structured object

    return "undefined"

  $\forall \langle u,\sigma,v\rangle \in y$ do                                     for each triple in y in order do

    if $\sigma \notin \mathcal{R}_x$ then                                 relation must be defined for structure

      return "undefined"

    if not($M_x$) and $\exists w \ni \rho_2(u)=\rho_2(w)$ then        if there is no replication and value of

      $u \leftarrow w$                                        u already in structure use object
                                                     already in structure

    if not($M_x$) and $\exists w \ni \rho_2(v)=\rho_2(w)$ then        same for v

      $v \leftarrow w$

    case of

     $u \notin \mathcal{O}_x \wedge v \notin \mathcal{O}_x$:                               neither u nor v already in structure

      if u=v then                                    if u and v the same object then if

        if (($\sigma$ 1-many and $\sigma$ unique range) or      relation is 1-many unique range or

           ($\sigma$ many-1 and $\sigma$ unique domain))        many-1 unique domain and connected then
        and $\sigma$ connected                         operation is not defined
    then return "undefined"
    else                                               otherwise, if structure is empty and
      if $|\mathcal{O}_x|=0$ and ($\sigma$ unique                 unique domain or unique range then
         domain or $\sigma$ unique range)                   operation is not defined,
        then return "undefined"
        else                                           else relate u and v if possible
          if def(u,$\sigma$,v)
            then exitcase
            else return "undefined"
if ($\sigma$ total domain or $\sigma$ total range)            relation cannot be total domain or range
   and not($\sigma$ symmetric or                          and symmetric or reflexive
   $\sigma$ reflexive)
  then return "undefined"
if $|\mathcal{O}_x|=0$                                          if structure empty is u a head or v a
  then                                                 tail?
    if $\sigma$ unique range then $h_\sigma \leftarrow u$
    if $\sigma$ unique domain then $t_\sigma \leftarrow v$
    else                                    else if structure not empty, if
      if σ connected                        connected then op not defined, otherwise
        then return "undefined"             if not unique domain or range op is ok
        else
          if σ unique domain or
             σ unique range then
           return "undefined"
  if not(def(u,σ,v)) then                   relate u to v if possible
    return "undefined"
  if σ reflexive then                       if relation reflexive relate u to u and
    if def(u,σ,v)                           v to v
      then
        if not(def(v,σ,v)) then
          return "undefined"
      else return "undefined"
  if σ symmetric then                       if relation symmetric relate v to u
    if not(def(v,σ,u)) then
      return "undefined"
```

```
exitcase                                        leave case statement
$u \in \Theta_x \wedge v \notin \Theta_x$:      u is and v is not in structure
case of
 $\sigma$ 1-1:                                  relation is 1 to 1
   if $\exists\langle\langle u,w\rangle,\delta\rangle \in \sigma$ then      if u related to something (w), then
     if undef(u,$\sigma$,w)                     unrelate u to w and relate u to v and
       then                                     v to w
         if $\sigma$ connected or               this is allowed only if connected or
            not($\sigma$ partial domain and     1-1 partial domain partial range conn.
            $\sigma$ partial range)
         then
            if not(def(v,$\sigma$,w)) then
              return "undefined"
       else return "undefined"
   if $\sigma$ unique domain and $t_\sigma$=u then      v is now tail if u previously was a tail
     $t_\sigma \leftarrow v$
   exitcase                                     leave case statement
 $\sigma$ 1-many:                               relation is 1 to many
```

```
  exitcase                                   no problems
 $\sigma$ many-1:                            relation is many to 1
  if $\exists$<<u,w>,$\delta$>$\in\sigma$ then            if u related to something (w) then
    if undef(u,$\sigma$,w)                   unrelate u to w and relate u to v and
      then                                   v to w
        if not(def(v,$\sigma$,w)) then
          return "undefined"
      else return "undefined"
  if $\sigma$ unique domain and $t_\sigma$=u then       make v tail is u previously was a tail
    $t_\sigma \leftarrow$v
  exitcase                                   leave case statement
 $\sigma$ many-many:                         relation is many to many
  if $\sigma$ total domain and               if relation is total domain and not
     not($\sigma$ symmetric or $\sigma$ reflexive)    reflexive or symmetric then operation
    then return "undefined"                  is not defined
  if $\sigma$ unique domain then             if unique domain and u not tail,
    if $t_\sigma$=u                          operation is not defined, otherwise v is
      then $t_\sigma \leftarrow$v            new tail
```

```
        else return "undefined"
      if $\sigma$ reflexive then                    if reflexive relate v to v
        if not(def(v,$\sigma$,v)) then
          return "undefined"
      if $\sigma$ symmetric then                    if symmetric relate v to u
        if not(def(v,$\sigma$,u)) then
          return "undefined"
      exitcase                                      leave case statement
    endcase                                         end of nested case
    if not(def(u,$\sigma$,v))                       relate u to v, if possible
      then return "undefined"
    exitcase                                        leave outer case statement
   $u \notin \mathcal{O}_x \wedge v \in \mathcal{O}_x$:                     u is not and v is in structure
    case of
     $\sigma$ 1-1:                                  relation is one to one
      if $\exists$<<w,v>,$\mathcal{S}$>$\in \sigma$ then               if v is in relation to something (w)
        if undef(w,$\sigma$,v)                      then unrelate w to v and relate w to u
          then                                      and u to v (must not be connected or
            if σ connected or                      not partial domain and partial range
               not(σ partial domain and
               σ partial range)
              then
                if not(def(w,σ,u)) then
                  return "undefined"
          else return "undefined"
      if σ unique range and $h_\sigma$=v then      make u head if v was previously a head
        $h_\sigma \leftarrow$ u
      exitcase                                     leave case statement
    σ 1-many:                                      relation is 1 to many
      if $\exists$<<w,v>,$\delta$>$\in$σ then     if v in relation to something (w), then
        if undef(w,σ,v)                            unrelate w to v and relate w to u and
          then                                     u to v
            if not(def(w,σ,u)) then
              return "undefined"
          else return "undefined"
      if σ unique range and $h_\sigma$=v          make u head if v was previously
      then $h_\sigma \leftarrow u$
    exitcase                                   leave case statement
   $\sigma$ many-1:                            relation is many to 1
    exitcase                                   just relate u to v
   $\sigma$ many-many:                         relation is many to many
    if $\sigma$ total range and                if relation is total range and not
       not($\sigma$ symmetric or $\sigma$ reflexive)   symmetric or reflexive then operation is
     then return "undefined"                   not defined
    if $\sigma$ unique range then              if unique range and v  head then
      if $h_\sigma = v$                        make u new head, else operation not
        then $h_\sigma \leftarrow u$           defined
        else return "undefined"
    if $\sigma$ reflexive then                 if relation reflexive then relate u to u
      if not(def(u,$\sigma$,u)) then
        return "undefined"
    if $\sigma$ symmetric then                 if relation symmetric then relate v to u
      if not(def(v,$\sigma$,u)) then
        return "undefined"
     exitcase                                          leave case statement
   endcase                                             end of nested case statement
   if not(def(u,$\sigma$,v))                           relate u to v
     then return "undefined"
   exitcase                                            leave outer case statement
 $u\in\theta_x \wedge v\in\theta_x$:                   u and v are in structure
   if $\exists\langle\langle u,v\rangle,\delta\rangle\in\sigma$ then exitcase        if u already related to v leave case
                                                       statement
   case of
    $\sigma$ 1-1:                                      relation is 1 to 1
     return "undefined"                                operation not defined
    $\sigma$ 1-many:                                   relation is 1 to many
     if $\exists\langle\langle w,v\rangle,\delta\rangle\in\sigma$ then              if there is something related to v (w),
       if undef(w,$\sigma$,v)                          then unrelate v to w and relate u to v,
         then                                          must check that structure is still
           if notconn($\theta_x$,$\sigma$) then        connected with respect to relation
             return "undefined"
         else return "undefined"
     if $\sigma$ unique range then                     if unique range and u=v then operation
```

```
      if u=v                                   not defined, and if not u=v, then look
        then return "undefined"                for something (w) related to u, unrelate
        else                                   w to u and relate u to v
          if ∃<<w,u>,δ>∈σ then
            if not(undef(w,σ,u)) then
              return "undefined"
            hσ← u                              u is new head
    exitcase                                   leave case statement
  σ many-1:                                    relation is many to 1
    if ∃<<u,w>,δ>∈σ then                       if u is related to something (w), then
      if undef(u,σ,w)                          unrelate u to w and relate u to v,
        then                                   must check that structure is still
          if notconn(σx,σ) then                connected with respect to the relation
            return "undefined"
        else return "undefined"
    if σ unique domain then                    if unique domain and u=v then operation
      if u=v                                   not defined, and if not u=v then look
        then return "undefined"                for v related to something and unrelate
        else                                        it
          if $\exists\langle\langle v,w\rangle,\mathscr{S}\rangle\in\sigma$ then
            if not(undef(v,$\sigma$,w)) then
              return "undefined"
            $t_\sigma \leftarrow v$                            make v new tail
    exitcase                                        leave case statement
   $\sigma$ many-many:                                 relation is many to many
    if $\sigma$ unique range and $h_\sigma=v$               if destroying head, then operator is not
      then return "undefined"                       defined
    if $\sigma$ unique domain and $t_\sigma=u$              if destroying tail, then operator is not
      then return "undefined"                       defined
    if $\sigma$ symmetric then                          if symmetric then relate v to u
      if not(def(v,$\sigma$,u)) then
        return "undefined"
    exitcase                                        leave case statement
   endcase                                          end of nested case statement
   if not(def(u,$\sigma$,v))                            relate u to v if possible
     then return "undefined"
```

```
      exitcase                                leave case statement
    endcase                                   end of case statement
    if $u \notin \Theta_x$ then               if u not in structure, then insert it
      $\Theta_x \leftarrow \Theta_x \cup u$
      if $P_x$ defined then order(u,x)        order u in structure if necessary
    if $v \notin \Theta_x$ then               same for v
      $\Theta_x \leftarrow \Theta_x \cup v$
      if $P_x$ defined then order(v,x)
  $\forall \sigma \in R_x$ do                 check that structure is legal with
    if not(legal(x,$\sigma$)) then            respect to each relation
      return "undefined"
  return "defined"                            operation is defined
end
```

UNRELATE

```
unrelate(x,y)                                      in structure x unrelate <obj,rel,obj>
                                                   triples in y

  if $x \in \mathcal{V}_p$ then return "undefined"               structure must not be primitive
  $\forall \langle u,\sigma,v\rangle \in y$ do                            for all triples in y in turn do
    if $u \notin \mathcal{O}_x$ or $v \notin \mathcal{O}_x$ or $\sigma \notin \mathcal{R}_x$                objects must be in structure and
      then return "undefined"                      relation defined for structure
    if $\exists\langle\langle u,v\rangle,\delta\rangle \in \sigma$ then                          if u related to v do
      if not($\sigma$ connected or $\sigma$ many-many             must be connected, many-many or
         or ($\sigma$ partial domain and                   partial domain and partial range
         $\sigma$ partial range))
        then return "undefined"
      if $\exists\langle\langle u,w\rangle,\delta'\rangle \in \sigma$ and ($\sigma$ total domain       if u related to something else and total
         or ($\sigma$ unique domain and $t_\sigma \neq u$))          domain or (unique domain and u not tail)
        then return "undefined"                    then op not defined
      if $\exists\langle\langle w,v\rangle,\delta'\rangle \in \sigma$ and ($\sigma$ total range        if something related to v and total
         or ($\sigma$ unique range and $h_\sigma \neq v$))           range or (unique range and v not head)
        then return "undefined"                    then op not defined
      if $\sigma$ reflexive and u=v                       cannot unrelate reflexive relation, u to
        then return "undefined"                          u
      $\sigma \leftarrow \sigma \setminus \langle\langle u,v\rangle,\ell\rangle$                 unrelate u and v
      if $\sigma$ symmetric then                          if symmetric unrelate v and u
        $\sigma \leftarrow \sigma \setminus \langle\langle v,u\rangle,\ell\rangle$
      if $\sigma$ connected then                          if connected relation check that
        if notconn($\Theta_x$,$\sigma$) then                  structure is still connected
          return "undefined"
  return "defined"                                     operation is defined
end
```

RELATED

```
related($x,\sigma,y$)                                 return all elements related to x
  if $y \in \Upsilon_p$ or $x \notin \Theta_y$ or $\sigma \notin \mathcal{R}_y$ then        if y is a primitive structure or x is
    return "undefined"                              not an element of or $\sigma$ is not a relation
                                                    for y then the operation is not defined
  return $\{u \in \Theta_y \mid \exists\langle\langle x,u\rangle,\delta\rangle \in \sigma\}$                return the related elements
end
```

READADTTR

```
readattr(x,<u,σ,v>,y)                          read the attribute y of the relation
                                               <u,σ,v> in structure x

  if $x \in \Upsilon_p$ or $u \notin \Theta_x$ or $v \notin \Theta_x$ or        if x is a primitive structure or u or v
      $\sigma \notin \mathcal{R}_x$ or $\nexists \langle\langle u,v\rangle,\mathcal{S}\rangle \in \sigma$ then    is not an element of x or σ is not a
    return "undefined"                         relation on x or there does not exist
                                               σ relation between u and v then the
                                               operation is not defined

  if $\exists \langle y,z\rangle \in \mathcal{S}$
    then return z                              if the attribute exists return it
    else return "undefined"                    otherwise the operation is not defined
end
```

STOREATTR

```
storeattr(x,<u,$\sigma$,v>,<y,z>)                       store attribute <y,w> for relation
                                                        <u,$\sigma$,v> in structure x

  if $x \in \Upsilon_p$ or $u \notin \mathcal{O}_x$ or $v \notin \mathcal{O}_x$ or      if x is a primitive structure or u or v
      $\sigma \notin \mathcal{R}_x$ or $\nexists \langle\langle u,v\rangle,\mathcal{S}\rangle \in \sigma$ then          is not an element of x or $\sigma$ is not a
    return "undefined"                                  relation on x or the relation u to v is
                                                        not defined then the operation is not
                                                        defined.

  if $\exists \langle y,w\rangle \in \mathcal{S}$ then                            if the y attribute is currently
    $\mathcal{S} \leftarrow \mathcal{S} \setminus \langle y,w\rangle$                                defined then remove it
  $\mathcal{S} \leftarrow \mathcal{S} \cup \langle y,z\rangle$                                   add the attribute <y,z> to the relation
  if $\sigma$ symmetric then                            if the relation is symmetric must also
    if $\exists \langle\langle v,u\rangle,\mathcal{S}\rangle \in \sigma$ then                  define attribute for symmetric relation
      $\mathcal{S} \leftarrow \mathcal{S} \setminus \langle y,u\rangle$                              value
      $\mathcal{S} \leftarrow \mathcal{S} \cup \langle y,z\rangle$
  return "defined"                                      operation is defined
end
```

ASSIGN

```
assign(x,y)                                 assign a copy of the value of y to x

  if "P defined in terms of x" then         may not assign to order field of the

    return "undefined"                      structure (it would result in an
                                            implicit reordering)

  storecopy(y,x)                            store a copy of the value of y in the
                                            value of x

  return "defined"                          operation is defined

end
```