stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 158/81 JANUARI

H.B.M. JONKERS

# ABSTRACT STORAGE STRUCTURES

Preprint

kruislaan 413 1098 SJ amsterdam

*Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.*

1980 Mathematics subject classification: 68B15

ACM-Computing Reviews-category: 4.34

# Abstract storage structures[*)]

by

H.B.M. Jonkers

ABSTRACT

A novel model for the description of storage structures is presented. It is based on the consideration that a storage structure is completely characterized by two things: the collection of its access paths and a relation which indicates whether two access paths access the same substructure. The model, called a "structure", is abstract in the sense that it is free of low level concepts such as pointers and garbage, while at the same time it is general in that it allows the description of storage structures with arbitrary sharing and circularities. Operations on structures (such as creation and replacement) can be described very naturally in terms of three primitive operations. These primitive operations are defined using a special partial order, which turns the set of all structures into a complete lattice.

KEY WORDS & PHRASES: storage structure, data structure, path, object, sharing, circularity

[*)] This report will be submitted for publication elsewhere.

## 1. INTRODUCTION

The question what a "data structure" is has been a point of dispute for several years. Though not all powder smoke has drifted away yet, a beginning of agreement can now be observed. A data structure is a class of objects which is fully characterized by the operations which can be applied to those objects. There are two aspects to this characterization: an external and an internal aspect. The external aspect deals with the question *what* the effect of the operations is. The concept of an "abstract data type" [11], which is essentially a heterogeneous algebra [3], has been introduced to model this aspect of a data structure. The internal aspect is concerned with the question *how* the effect of the operations is accomplished. This aspect is usually dealt with by choosing a "representation" for the data structure and "implementing" each operation in terms of the well-known operations on the representation. It is generally agreed that the internal aspect of a data structure should be hidden ("encapsulated" [16]) to the user.

The above agreement on what a data structure is does not carry over to an other crucial question: How should data structures be described, or "specified"? It is important, both to the user and to the implementer, that a specification of a data structure describes only the external aspect of the data structure. The meaning (in the semantical sense) of a specification of a data structure must therefore be an abstract data type. There are basically two ways to specify data structures (or abstract data types, if you like) [12].

The first, and apparently the most attractive, is the axiomatic (or "implicit") method [6, 7]. In this method the essential properties of the operations are described through axioms. The major advantage of this method is that it is not necessary to commit oneself to a representation for the data structure. There are also two severe drawbacks, however. Apart from very simple data structures, it is very difficult to construct complete and consistent axiomatic specifications. Specifically data structures involving "dynamic" and "shared" data, which are frequently encountered in practice, are very hard to specify. Moreover, axiomatic specifications are usually far from easy to comprehend.

The second way of specifying abstract data types is the "abstract model" approach [1]. In this approach an abstract representation for the data structure to be specified is chosen. The operations of the data structure are then specified in terms of this representation. This method clearly contrasts the axiomatic method as to its advantages and disadvantages. First of all, specifications are more easily constructed. If the possibility of dynamic creation and sharing is already included in the abstract representations chosen, data structures featuring these properties are readily specified. The specifications also tend to be more readable than axiomatic specifications. The salient disadvantage, of course, is the fact that specifications are not representation-independent. If one is not very careful, details of the representation chosen may permeate into the external world and lead to an "overspecification" of the data structure. (Contrast this with the problem of writing complete axiomatic specifications.)

It is my firm belief that for realistic applications the future lies in the abstract model approach. A precondition is, however, that the problem of representation-dependence is solved satisfactorily. The key to a solution of this problem lies in the observation that the choice of a representation need not depend on efficiency considerations. The only criteria in choosing a representation should be the clarity and naturalness of the specification. This implies first of all that the representations themselves must be free of implementation detail, or in other words, they should be as abstract as possible. In particular they should not include such things as pointers, fixed size storage cells, etc.. On the other hand, the possibility of dynamic creation and sharing should be inherent (otherwise many applications are ruled out). If we had such abstract representations at our disposal, data structures could be specified relatively representation-independent. The sole purpose of the representation would be to increase the comprehensibility of the specification, and not to suggest a certain implementation.

In this paper representations will be described which are believed to satisfy the requirements mentioned above. These representations can be viewed as abstract "storage structures". They can be used as the basis for a specification method, which allows the specification of realistic data structures in a comprehensible and unambiguous way, without undue effort and at various levels of abstraction. Their use is not restricted to specification languages, however. It is envisaged that they can successfully be used in definitions of programming languages as well, especially in definitions of those programming languages which feature sharing ("aliasing") and dynamic creation of data.

The representations, which will be called "structures", are introduced in the next section, together with some related concepts. In Section 3 three primitive operations which can be applied to structures are defined. For their definition a partial order, which turns the set of all structures into a complete lattice, is introduced first.

## 2. STRUCTURES

The purpose of this section is to define the concept of a "structure". A structure can be viewed as an abstract "storage structure", which can be "accessed" through special keys called "accessors". Accessors will be considered as primitive concepts, usually denoted by strings of letters and digits. By repeatedly applying accessors to a structure one can follow an "access path".

An *accessor* is a primitive concept.

$\mathcal{A}$ is the set of all accessors.

$\mathcal{A}^*$ is the set of all finite sequences of accessors.

$\mathcal{A}^+$ is the set of all finite nonempty sequences of accessors

$\Lambda$ is the empty sequence of accessors.

The sequence $A_1, \ldots, A_n$ of accessors will be denoted as $A_1 \ldots A_n$.

The following definition of the concept of a structure is based on the consideration that a (storage) structure is completely characterized by two things: First, the collection of all of its access paths and second, a relation which indicates whether two access paths access the same "substructure". (Notice that the latter is necessarily an equivalence relation.) Taking into account the properties of access paths as well we arrive at the following definition:

A *structure* S is a pair $<\mathcal{P}, \equiv>$, where $\mathcal{P} \subset \mathcal{A}^*$ and $\equiv$ is an equivalence relation on $\mathcal{P}$ such that

1. $\Lambda \in \mathcal{P}$.
2. $PA \in \mathcal{P} \Rightarrow P \in \mathcal{P}$. $\qquad (P \in \mathcal{A}^*, A \in \mathcal{A})$
3. $PA \in \mathcal{P} \wedge P \equiv Q \Rightarrow QA \in \mathcal{P} \wedge PA \equiv QA$. $\qquad (P, Q \in \mathcal{P}, A \in \mathcal{A})$

A $P \in \mathcal{P}$ will be called a *path* of S.

An $X \in \mathcal{P}|\equiv$, i.e. an equivalence class of $\equiv$, will be called an *object* of S.

$\mathcal{S}$ is the set of all structures.

Property 1 states that the empty sequence of accessors is a path of S (hence $\mathcal{P} \neq \phi$). Property 2 implies that any head piece of a path of S is also a path of S. Property 3 states that equivalent paths have equivalent continuations. This property of an equivalence relation is known as "right-invariance". The paths of a structure can be viewed as "names" for the objects which they represent. As will be seen later, the concept of an object as introduced above is closely related to the intuitive concept of an object.

There are three trivial examples of a structure, which will be called the "empty structure", the "convergent structure" and the "divergent structure" respectively:

$\perp = \langle\{\Lambda\}, \{(\Lambda, \Lambda)\}\rangle$ is a structure called the *empty structure.*

$T_C = \langle A^*, A^* \times A^*\rangle$ is a structure called the *convergent structure.*

$T_D = \langle A^*, \{(P, P) \mid P \in A^*\}\rangle$ is a structure called the *divergent structure.*

Notice that $\perp$ and $T_C$ contain only a single object, while $T_D$ contains an infinite number of objects (i.e. if $A \neq \phi$, which we will from now on assume). Other examples of structures will be discussed below.

*Example 1*

Let $S = \langle P, \equiv\rangle$, where

$P = \{\Lambda, a, b, ba\}$,

$\equiv\; = \{(\Lambda, \Lambda), (a, a), (a, ba), (ba, a), (ba, ba), (b, b)\}$,

then S is a structure containing the following objects:

$P|\equiv\; = \{\{\Lambda\}, \{a, ba\}, \{b\}\}$.

Notice that the paths a and ba are "aliases" for one and the same object.

*End of Example*

Before continuing some notations have to be introduced. First, if $S = \langle P, \equiv\rangle$ is a structure, then $P_S$ and $\equiv_S$ will denote $P$ and $\equiv$ respectively. Second, if X is an object of a structure S and P is a path of S such that $P \in X$, then, if no confusion can arise, $\overline{P}$ will denote X. This convention fits in with the common mathematical practice of denoting equivalence classes by their representatives. Definitions and lemmas which use this notation for objects must be proved to be independent of the choice of the representatives for the objects.

The definition of a structure does not preclude that structures use an infinite number of accessors or have an infinite number of objects. Structures that use only a finite number of accessors and have a finite number of objects constitute an important subclass. The structures in this subclass will be called the "finite structures".

Let S be a structure.

The *accessor set* of S is defined as:
$\{A \in \mathcal{A} \mid \exists\, P \in \mathcal{P}_S\ [PA \in \mathcal{P}_S]\}$.

S is called *finite* iff the accessor set and the set of objects of S are finite; otherwise S is called *infinite*.

The empty structure $\perp$ is an example of a finite structure, and the divergent structure $\top_D$ is an example of an infinite structure. The convergent structure $\top_C$ is infinite if and only if $\mathcal{A}$ is infinite.

Finite structures can be pictured in a systematic way as follows:

```
For each object $\overline{P}$
    Draw a circle $\mathcal{C}_{\overline{P}}$.
For each pair of objects $(\overline{P}, \overline{Q})$
and each accessor A with $PA \in \overline{Q}$
    Draw an arrow labeled by A from $\mathcal{C}_{\overline{P}}$ to $\mathcal{C}_{\overline{Q}}$.
Label $\mathcal{C}_{\overline{\Lambda}}$ by $\Lambda$.
```

Notice that this drawing algorithm is independent of the choice of the paths for the objects and that it would never terminate if applied to an infinite structure. It is easy to see that the picture thus associated to a finite structure is unique.

*Example 2*

The empty structure $\perp$ has the following picture:

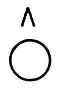


Fig. 1

If $A$ = {a, b}, then the picture of the convergent structure $T_C$ is:

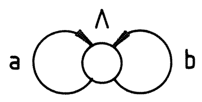


Fig. 2

If we try the impossible and apply the drawing algorithm to the divergent structure $T_D$ with $A$ = {a, b}, then we get:

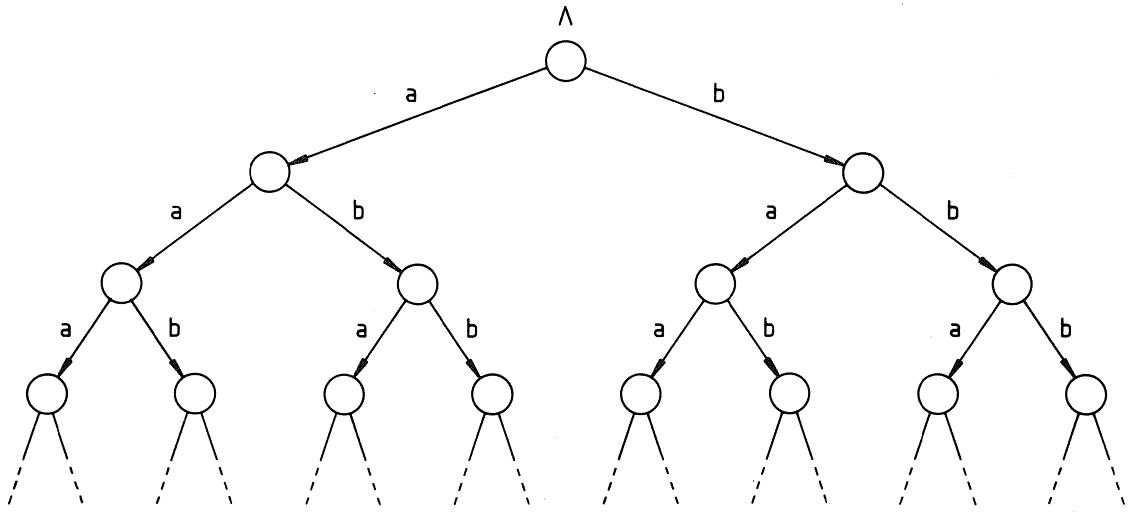


Fig. 3

The picture of the structure S from Example 1 is:

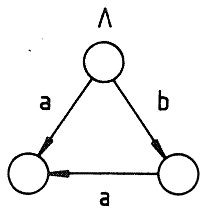


Fig. 4

*End of Example*

The above may raise the question what the difference is between a structure and a rooted graph with labeled edges. At first sight there may not seem to be any difference, yet there is. There are two crucial differences. First, the concept of "unreachability" is meaningless in a structure. Each object has at least one access path. Second, objects do not have a separate identity. An object simply *is* the collection of its access paths. These two facts will be seen to have a number of important consequences.

An other important observation is that the paths of a structure should *not* be considered as "pointers": Though a path can be viewed as a name for an object, paths are not objects themselves. Instead, the arrows in the picture of a structure should be regarded as denoting physical inclusion. Since arbitrary kinds of physical inclusion (such as sharing and circularity) can be modeled in a structure, the need to introduce pointers will nowhere arise. The concept of physical inclusion will be made more precise by introducing three relations on the set of objects of a structure:

Let S be a structure.
Let $\overline{P}$ and $\overline{Q}$ be objects of S.

$\overline{P}$ is a *direct component* of $\overline{Q}$ iff there is an $A \in \mathcal{A}$ such that $QA \in \overline{P}$.

$\overline{P}$ is a *component* of $\overline{Q}$ iff there is an $R \in \mathcal{A}^+$ such that $QR \in \overline{P}$.

$\overline{P}$ is *contained* in $\overline{Q}$ iff there is an $R \in \mathcal{A}^*$ such that $QR \in \overline{P}$.

Check that these definitions are independent of the choice of P and Q. The relations "be a component of" and "be contained in" are both transitive, while the latter is also reflexive. Neither of them need be an (irreflexive or reflexive) partial order (see Example 3). The meaning of the fact that an object is "cyclic" can be defined as follows:

An object of a structure is *cyclic* iff it is a component of itself.

It is easy to see that cyclic objects contain an infinite number of paths.

*Example 3*

Consider the structure S of Figure 5.

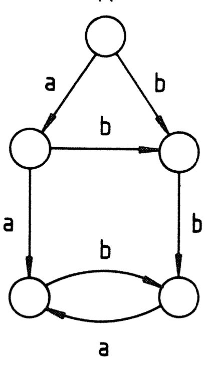


Fig. 5

The objects of S are:

$$\overline{\Lambda} = \{\Lambda\},$$
$$\overline{a} = \{a\},$$
$$\overline{b} = \{ab, b\},$$
$$\overline{aa} = \{P(ba)^n \mid n \geq 0 \wedge P \in \{aa, abba, bba\}\},$$
$$\overline{bb} = \{P(ab)^n \mid n \geq 0 \wedge P \in \{aab, abb, bb\}\}.$$

The three inclusion relations which are defined between these objects can be described schematically as follows (the plus sign indicates where the relation holds):

$\overline{P}$ is a direct component of $\overline{Q}$:

| $\overline{P} \setminus \overline{Q}$ | $\overline{\Lambda}$ | $\overline{a}$ | $\overline{b}$ | $\overline{aa}$ | $\overline{bb}$ |
|---|---|---|---|---|---|
| $\overline{\Lambda}$ | - | - | - | - | - |
| $\overline{a}$ | + | - | - | - | - |
| $\overline{b}$ | + | + | - | - | - |
| $\overline{aa}$ | - | + | - | - | + |
| $\overline{bb}$ | - | - | + | + | - |

$\overline{P}$ is a component of $\overline{Q}$:

| $\overline{P} \setminus \overline{Q}$ | $\overline{\Lambda}$ | $\overline{a}$ | $\overline{b}$ | $\overline{aa}$ | $\overline{bb}$ |
|---|---|---|---|---|---|
| $\overline{\Lambda}$ | - | - | - | - | - |
| $\overline{a}$ | + | - | - | - | - |
| $\overline{b}$ | + | + | - | - | - |
| $\overline{aa}$ | + | + | + | + | + |
| $\overline{bb}$ | + | + | + | + | + |

$\overline{P}$ is contained in $\overline{Q}$:

| $\overline{P} \setminus \overline{Q}$ | $\overline{\Lambda}$ | $\overline{a}$ | $\overline{b}$ | $\overline{aa}$ | $\overline{bb}$ |
|---|---|---|---|---|---|
| $\overline{\Lambda}$ | + | - | - | - | - |
| $\overline{a}$ | + | + | - | - | - |
| $\overline{b}$ | + | + | + | - | - |
| $\overline{aa}$ | + | + | + | + | + |
| $\overline{bb}$ | + | + | + | + | + |

The relation "be a component of" is not an irreflexive partial order here, because it is not irreflexive: $\overline{aa}$ is a component of itself. The relation "be contained in" is not a reflexive partial order because it is not antisymmetric: $\overline{aa}$ is contained in $\overline{bb}$ and $\overline{bb}$ is contained in $\overline{aa}$, but $\overline{aa} \neq \overline{bb}$. This, of course, is caused by the fact that $\overline{aa}$ and $\overline{bb}$ are cyclic objects.

*End of Example*

The above example (and especially the expressions for the objects $\overline{aa}$ and $\overline{bb}$) suggests that there is a relation between structures and regular languages. Indeed, the objects of finite structures *are* regular languages:

<u>LEMMA 1</u>

Let S be a finite structure, then each object of S is a regular language over $A$.

This can be understood intuitively by considering the picture of a finite structure as the state diagram of a finite state machine and recalling the correspondence between finite state machines and regular languages. A straightforward proof can be obtained by using the fact that each equivalence class of a right-invariant equivalence relation of finite index is a regular language [8]. Another way to prove Lemma 1 is to use the relation between left-linear grammars and regular languages. (Check that a left-linear grammar, where each nonterminal symbol "produces" an object, can be associated to each structure.) Due to Lemma 1 a regular expression notation can now be used for the objects of all finite structures.

*Example 4*

The objects of the structures of Figures 1, 2, 4 and 5 can be denoted by regular expressions as follows:

Fig. 1: $\overline{\Lambda} = \Lambda$.

Fig. 2: $\overline{\Lambda} = (a+b)^*$.

Fig. 4: $\overline{\Lambda} = \Lambda$,
$\overline{a} = a+ba$,
$\overline{b} = b$.

Fig. 5: $\overline{\Lambda} = \Lambda$,
$\overline{a} = a$,
$\overline{b} = ab+b$,
$\overline{aa} = (aa+abba+bba)(ba)^*$,
$\overline{bb} = (aab+abb+bb)(ab)^*$.

*End of Example*

The concept of an object as we introduced it is closely related to the concept of a "dynamic object", as it is normally conceived in computer science. Dynamic objects are usually considered as "instances" of "values". Two dynamic objects may be instances of the same value and still be different. In mathematical models for dynamic objects this problem is usually solved by associating an "identity", which is an explicit value, to dynamic objects. As stated before, objects in structures do not have an explicit identity. It is interesting to see how the identity problem for them is solved. The objects in a structure can be viewed as instances of structures (so "structures" correspond to the "values" of dynamic objects). This is made more precise by the following definition of the "structure" of an object:

Let S be a structure.
Let $\overline{P}$ be an object of S.
The *structure* of $\overline{P}$, which will be denoted as $S[\overline{P}]$, is the structure T which is defined as follows:

$$P_T = \{Q \in A^* \mid PQ \in P_S\},$$

$$Q \equiv_T R \Leftrightarrow PQ \equiv_S PR. \qquad (Q, R \in P_T)$$

The proof that T is indeed a structure and that T is independent of the choice of P is simple. Two different objects can have the same structure (see Example 5). Hence they can be viewed as instances of that structure.

*Example 5*

Consider the structure S of Figure 6.

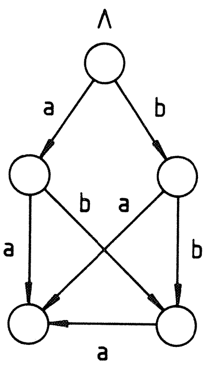


Fig. 6

In this figure we have (using regular expression notation):

$\overline{\Lambda} = \Lambda$,
$\overline{a} = a$,
$\overline{b} = b$,
$\overline{aa} = aa+aba+ba+bba$,
$\overline{bb} = ab+bb$.

The structure of $\overline{a}$ is:

$$S[\overline{a}] = \langle \mathcal{P}_0, \equiv_0 \rangle, \text{ where}$$

$$\mathcal{P}_0 = \{Q \in A^* \mid aQ \in \mathcal{P}_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_0 R \Leftrightarrow aQ \equiv_S aR, \qquad (Q, R \in \mathcal{P}_0)$$

hence $\mathcal{P}_0 | \equiv_0 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}$.
The structure of $\overline{b}$ is:

$$S[\overline{b}] = \langle \mathcal{P}_1, \equiv_1 \rangle, \text{ where}$$

$$\mathcal{P}_1 = \{Q \in A^* \mid bQ \in \mathcal{P}_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_1 R \Leftrightarrow bQ \equiv_S bR, \qquad (Q, R \in \mathcal{P}_1)$$

hence $\mathcal{P}_1 | \equiv_1 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}$.
So $\overline{a}$ and $\overline{b}$ have the same structure (the structure of Figure 4).

*End of Example*

*Example 6*

Consider the structure S of Figure 7.

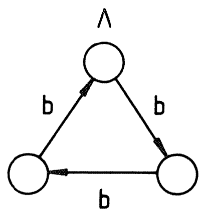


Fig. 7

All objects have the same structure:

$$S[\overline{\Lambda}] = S[\overline{b}] = S[\overline{bb}] = S.$$

*End of Example*

## 3. OPERATIONS ON STRUCTURES

In this section three primitive operations on structures will be defined. They constitute a sufficient set in the sense that all other useful operations on structures can be defined in terms of them. For their definition a special partial order on the set $\mathcal{S}$ of all structures will be introduced first.

The partial order $\sqsubset$ on $\mathcal{S}$ is defined as follows:

$$S \sqsubset T \Leftrightarrow \mathcal{P}_S \subset \mathcal{P}_T \wedge \equiv_S \subset \equiv_T. \qquad (S,\ T \in \mathcal{S})$$

The fact that $\sqsubset$ is indeed a partial order on $\mathcal{S}$ is trivial. In intuitive terms the fact that $S \sqsubset T$ means that S contains less paths than T and that in S less paths are "identified" than in T.

*Example 7*

The structures of Figure 8 form an ascending sequence:

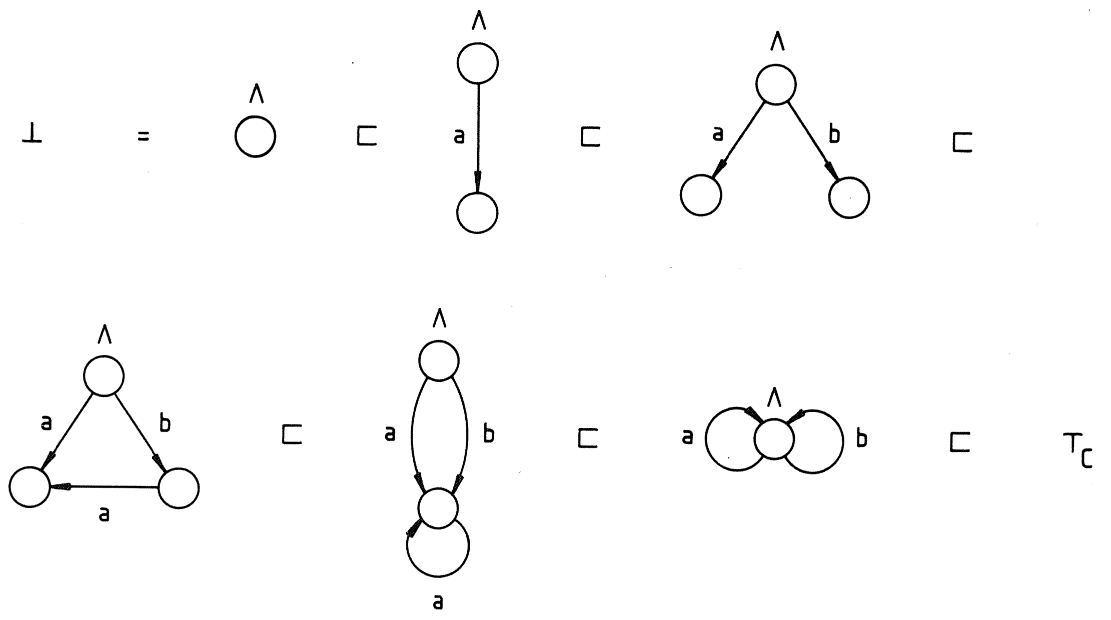


Fig. 8

*End of Example*

*Example 8*

If we define the partial order $\sqsubseteq_0$ on $\mathcal{S}$ as:

$$S \sqsubseteq_0 T \Leftrightarrow S \sqsubseteq T \wedge \mathcal{P}_S = \mathcal{P}_T, \qquad (S, T \in \mathcal{S})$$

then the fact that S $\sqsubseteq_0$ T means that S is a "partial expansion" of T, as illustrated in Figure 9.

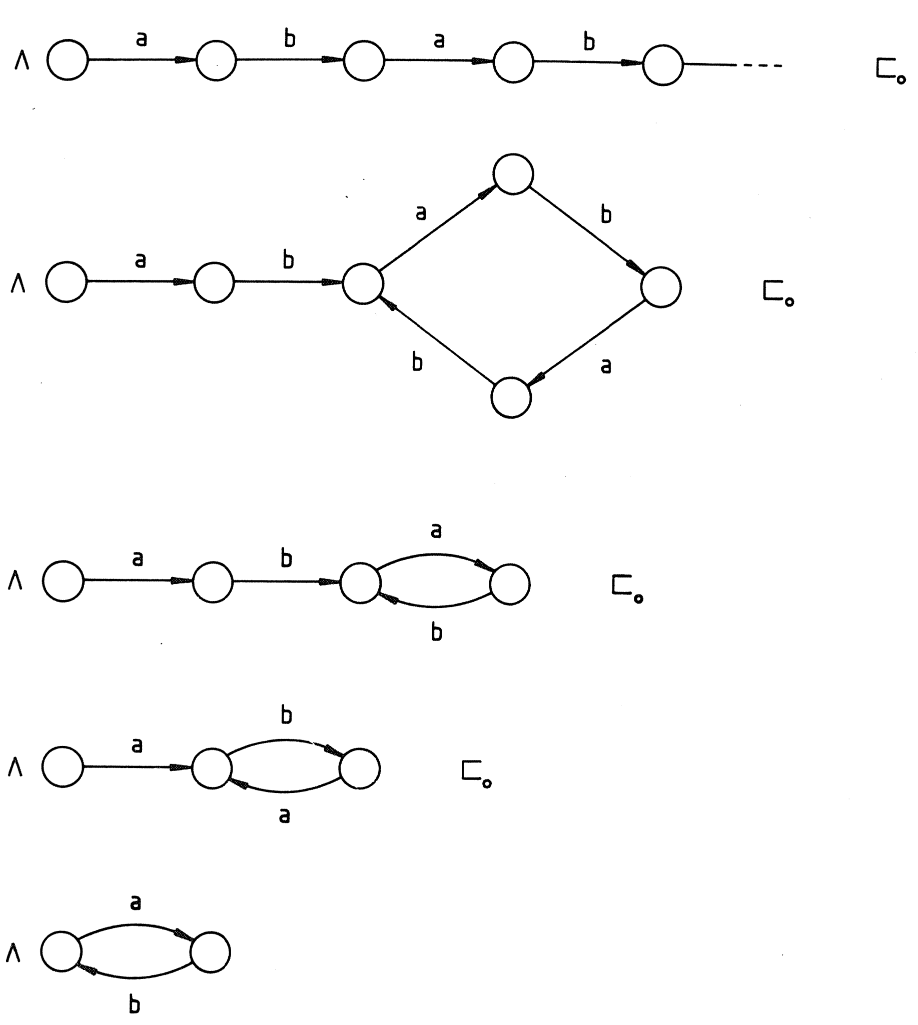


Fig. 9

*End of Example*

Notice that the partial orders $\sqsubset$ and $\sqsubset_0$ are much harder to describe in terms of graphs.

The relation $\sqsubset$ is more than just a partial order: It turns $S$ into a complete lattice. (A complete lattice is a partially ordered set where each subset has a greatest lower bound.) This is stated in:

<u>LEMMA 2</u>

$\langle S, \sqsubset \rangle$ is a complete lattice.

The proof of Lemma 2 is simple. First prove that, if S and T are structures, $<P_S \cap P_T, \equiv_S \cap \equiv_T>$ is also a structure. It is then easy to prove that the greatest lower bound of a set $T$ of structures is given by $<\cap_{T\in T} P_T, \cap_{T\in T} \equiv_T>$, where $\cap_{T\in T} P_T = A^*$ and $\cap_{T\in T} \equiv_T = A^* \times A^*$ if $T = \phi$. Notice that the empty structure $\perp$ and the convergent structure $\top_C$ are the "bottom" and "top" of the complete lattice $<S, \sqsubseteq>$, i.e. $\perp \sqsubseteq S \sqsubseteq \top_C$ for each $S \in S$. A simple theorem from lattice theory states that apart from a greatest lower bound, each subset also has a least upper bound [2]. The following definitions are therefore in order:

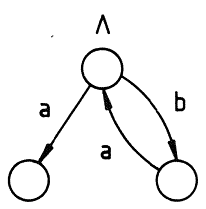
For each set $T$ of structures, the structures *inf* $T$ and *sup* $T$ are defined as follows:

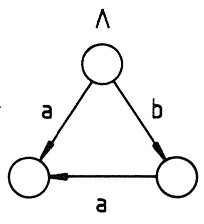
inf $T$ = greatest lower bound of $T$ with respect to $\sqsubseteq$,

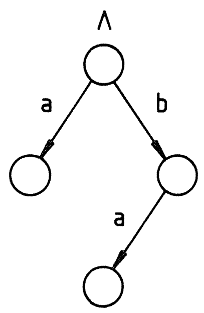
sup $T$ = least upper bound of $T$ with respect to $\sqsubseteq$.

The above will enable us to define the result of operations on structures in terms of inf's and sup's of arbitrary sets of structures without having to worry over the existence of the inf's and sup's.

*Example 9*

If S = 

and T = 

then inf{S, T} = 

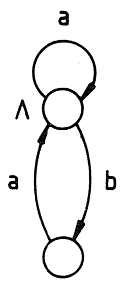
and sup{S, T} = 

Fig. 10

*End of Example*

Before defining the primitive operations on structures a remark should be made about an other interesting partial order on $\mathcal{S}$. The definition of $\sqsubseteq$ can be written as:

$$S \sqsubseteq T \Leftrightarrow \mathcal{P}_S \subset \mathcal{P}_T \wedge \forall\, P, Q \in \mathcal{P}_S\ [P \equiv_S Q \Rightarrow P \equiv_T Q]. \qquad (S, T \in \mathcal{S})$$

If we reverse the implication sign in this definition we still have a partial order, call it $\sqsubseteq_1$:

$$S \sqsubseteq_1 T \Leftrightarrow \mathcal{P}_S \subset \mathcal{P}_T \wedge \forall\, P, Q \in \mathcal{P}_S\ [P \equiv_T Q \Rightarrow P \equiv_S Q]. \qquad (S, T \in \mathcal{S})$$

Intuitively S $\sqsubseteq_1$ T means that S contains less paths than T and that in S less paths are "distinguished" than in T. The partial order $\sqsubseteq_1$ has both a bottom (the empty structure $\perp$) and a top (the divergent structure $\top_D$). Yet, in contrast with $\sqsubseteq$, it does not turn $\mathcal{S}$ into a complete lattice (see Example 10).

*Example 10*

Consider the structures in Figure 11.

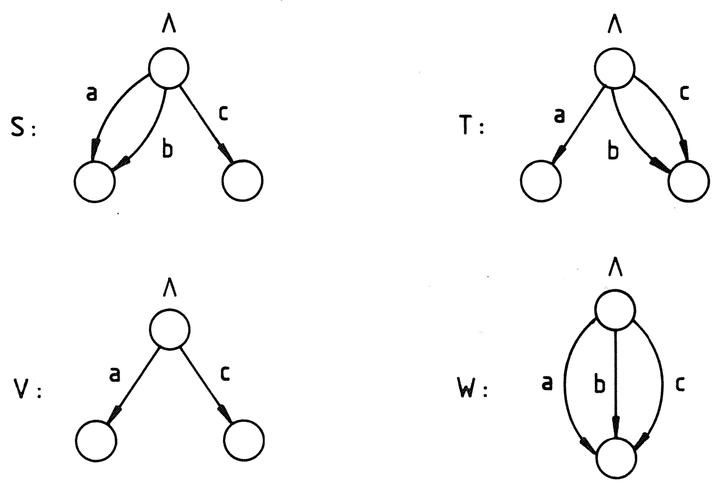


Fig. 11

Suppose S and T have a greatest lower bound X with respect to $\sqsubseteq_1$. Since V $\sqsubseteq_1$ S and V $\sqsubseteq_1$ T, we have that V $\sqsubseteq_1$ X. This implies that a, c $\in P_X$ and, since a $\not\equiv_V$ c, also that a $\not\equiv_X$ c. W $\sqsubseteq_1$ S and W $\sqsubseteq_1$ T imply that W $\sqsubseteq_1$ X, hence b $\in P_X$. X $\sqsubseteq_1$ S and a $\equiv_S$ b imply that a $\equiv_X$ b. Analogously, X $\sqsubseteq_1$ T and b $\equiv_T$ c imply that b $\equiv_X$ c. Using the transitivity of $\equiv_X$ we get a $\equiv_X$ c, which is a contradiction. Hence $<\mathcal{S}, \sqsubseteq_1>$ is not a complete lattice.

*End of Example*

All operations which will be introduced below are considered as partial operators on structures. They may have a number of parameters (usually objects in the structure to which they are applied, or accessors). The result of applying the operation F with parameters $X_1$, ..., $X_m$ to the structure S will be denoted as $\{S\}F(X_1, \ldots, X_m)$. The notation $F(X_1, \ldots, X_m)$ will be used to denote the (partial) operator $\lambda_{S\in\mathcal{S}}\ \{S\}F(X_1, \ldots, X_m)$. Concatenation is used to denote functional composition of operators, e.g. $F(X_1, \ldots, X_m)G(Y_1, \ldots, Y_n)$ denotes $\lambda_{S\in\mathcal{S}}\ \{\{S\}F(X_1, \ldots, X_m)\}G(Y_1, \ldots, Y_n)$.

The first primitive operation on structures which will be introduced amounts to the "creation" of an object in a structure. The created object has $\perp$ as its structure and is added as a direct component to a given object. The operation, called CRE, has two parameters $\overline{P}$ and A. $\overline{P}$ is an object in the structure S to which CRE is applied and A is an accessor such that PA is not a path of S. The effect of CRE($\overline{P}$, A) is pictured in Figure 12.

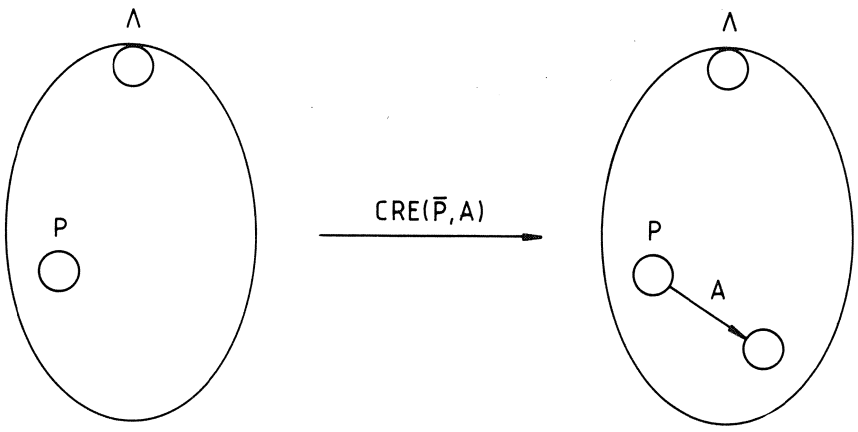


Fig. 12

The definition of CRE reads:

Let S be a structure. If $\overline{P}$ is an object of S and $A \in \mathcal{A}$ such that $PA \notin \mathcal{P}_S$, then $\{S\}CRE(\overline{P}, A)$ is the following structure:

$$\inf\{T \in \mathcal{S} \mid S \sqsubset T,\ \forall R \in \mathcal{P}_S\ [R \equiv_S P \Rightarrow RA \in \mathcal{P}_T]\}.$$

It should be clear that CRE($\overline{P}$, A) does what Figure 12 suggests. The fact that "less" in the partial order $\sqsubset$ implies "less identification" guarantees that a new object is created and not some old object is taken as the new component of $\overline{P}$.

*Example 11*

A binary tree can be generated from the empty structure by a sequence of operations such as:

$\{\bot\}\mathrm{CRE}(\overline{\Lambda}, a)\mathrm{CRE}(\overline{\Lambda}, b)\mathrm{CRE}(\overline{b}, a)\mathrm{CRE}(\overline{ba}, a)\mathrm{CRE}(\overline{ba}, b).$

The intermediate and final results of this sequence of operations are pictured in Figure 13.

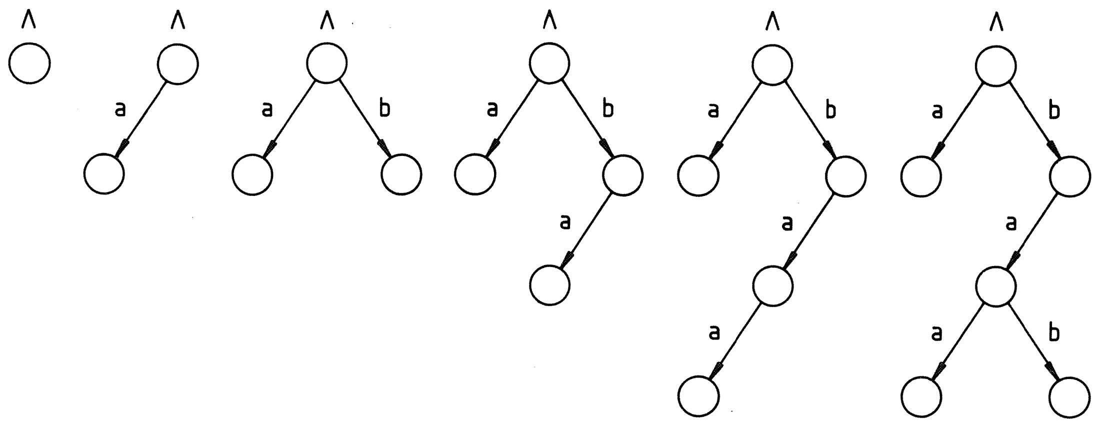


Fig. 13

*End of Example*

The second primitive operation on structures is like CRE, except that it adds an already existing object as a direct component to an object. The operation, called ADD, takes three parameters $\overline{P}$, A and $\overline{Q}$. $\overline{P}$ and $\overline{Q}$ are objects in the structure S to which ADD is applied and A is an accessor such that PA is not a path of S. The effect of ADD($\overline{P}$, A, $\overline{Q}$) is pictured in Figure 14.

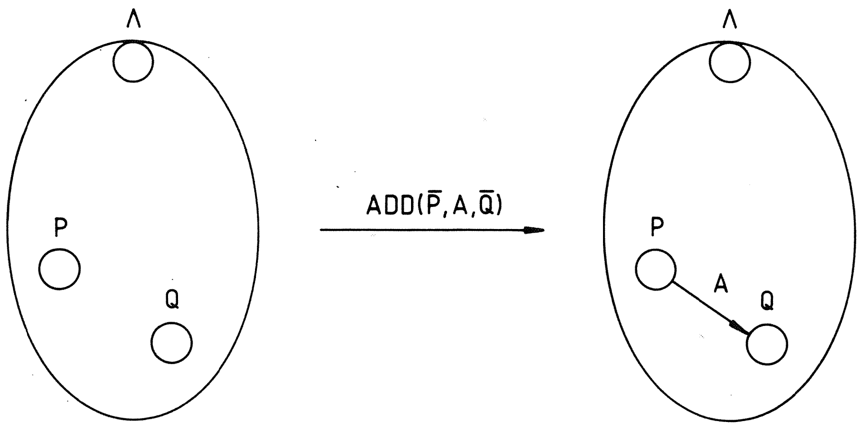


Fig. 14

The definition of ADD is given below:

Let S be a structure. If $\overline{P}$ and $\overline{Q}$ are objects of S and $A \in \mathcal{A}$ such that $PA \notin \mathcal{P}_S$, then $\{S\}ADD(\overline{P}, A, \overline{Q})$ is the following structure:

$$\inf\{T \in \mathcal{S} \mid S \sqsubset T,\ \forall R \in \mathcal{P}_S\ [R \equiv_S P \Rightarrow RA \in \mathcal{P}_T \wedge RA \equiv_T Q]\}.$$

The greatest lower bound of the same set of structures as in the definition of CRE is taken here, except that the set is restricted to those structures in which the paths RA with $R \equiv_S P$ and Q are identified. This guarantees that not a new object is created, but that $\overline{Q}$ is added as a new component to $\overline{P}$. Notice that, in contrast with CRE, it is not simple to define ADD without the use of the partial order $\sqsubset$. This is due to the fact that ADD may introduce circularities in a structure.

*Example 12*

Let S be the structure of Figure 15,

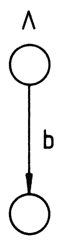


Fig. 15

then {S}ADD($\overline{b}$, a, $\overline{\Lambda}$) is the structure of Figure 16.

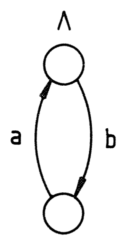


Fig. 16

*End of Example*

The third and final primitive operation can be viewed somehow as the (right) inverse of the other two primitive operations. It amounts to removing a direct component of an object. The operation, called REM, has two parameters $\overline{P}$ and A. $\overline{P}$ is an object in the structure S to which REM is applied and A is an accessor such that PA is a path of S. Figure 17 pictures the effect of REM($\overline{P}$, A).

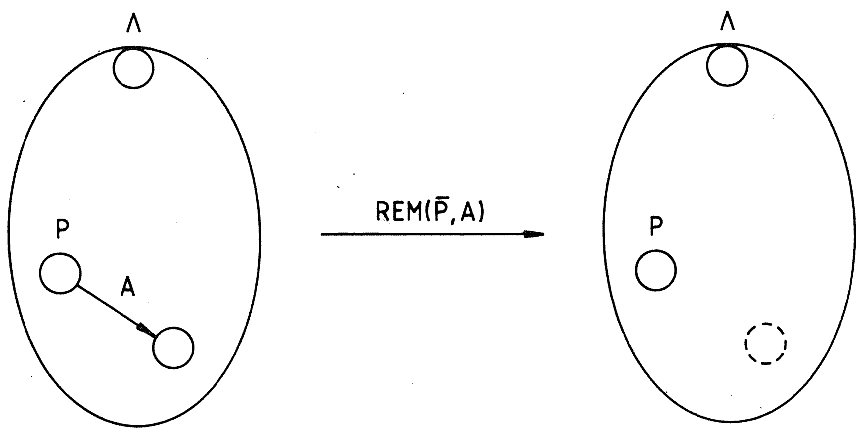


Fig. 17

The definition of REM is:

> Let S be a structure. If $\overline{P}$ is an object of S and $A \in \mathcal{A}$ such that $PA \in \mathcal{P}_S$, then $\{S\}REM(\overline{P}, A)$ is the following strcture:
>
> $$\sup\{T \in \mathcal{S} \mid T \sqsubset S, \forall R \in \mathcal{P}_S\ [R \equiv_S P \Rightarrow RA \notin \mathcal{P}_T]\}.$$

Notice that, due to the fact that objects may be shared, $REM(\overline{P}, A)$ need not remove the object $\overline{PA}$ from a structure. That is why this object is represented by a dotted circle in the right part of Figure 17. (Strictly speaking the path name P should also be dotted, because the path P (but not the object $\overline{P}$) may be removed from the structure by $REM(\overline{P}, A)$.) In general, $REM(\overline{P}, A)$ may reduce the number of objects in a structure by a number varying from zero to all but one (see Example 13).

*Example 13*

Consider the structure S of Figure 18.

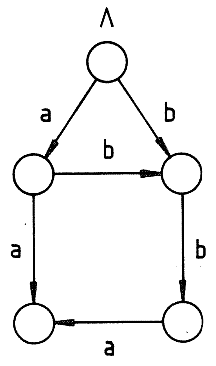


Fig. 18

The effect of REM($\overline{a}$, a) on S is:

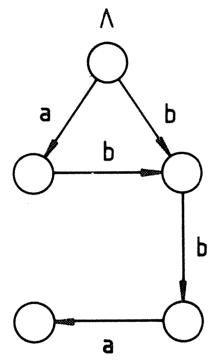


Fig. 19

Notice: the number of objects has not changed. If REM($\overline{ab}$, b) is applied subsequently to the structure of Figure 19, we get:

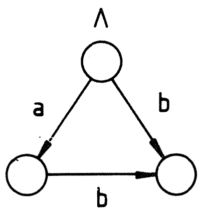


Fig. 20

Notice: two objects have "vanished".

*End of Example*

When choosing structures as the basis of the definition of a specification or programming language, the above three primitive operations are sufficient in the sense that all more complex operations can be expressed in terms of them. In order to illustrate this we shall sketch briefly how the meaning of language constructs can be described in terms of the primitive operations. The idea is to represent all values as structures (and their "instances" as objects of structures). If we consider the variables $X_1$, ..., $X_n$ of an algorithm as accessors, then the "state" of the algorithm can be represented by a structure as pictured in Figure 21.

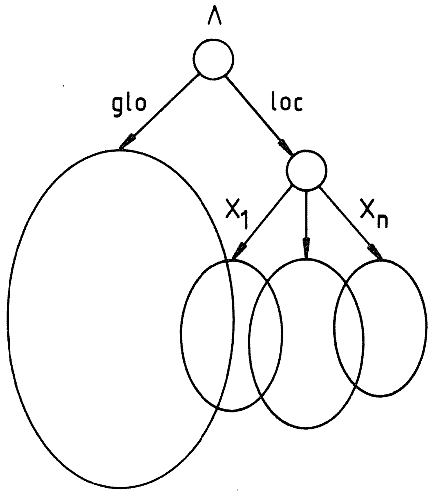


Fig. 21

In this figure the variables $X_1, \ldots, X_n$ of the algorithm are represented by the paths $loc.X_1, \ldots, loc.X_n$ (dots are used to separate accessors here). The values of the variables are (the structures of) the objects $\overline{loc.X_1}, \ldots, \overline{loc.X_n}$. Since the latter objects may share components, things such as "aliasing" can readily be described. The component $\overline{loc}$ of the state constitutes what might be called the "local environment". Apart from a local effect an algorithm may also have a global effect ("side effect"). This is modeled by the component $\overline{glo}$ (the "global environment") of the state, which is supposed to contain all information global to the algorithm. Since $\overline{glo}$ and $\overline{loc}$ may share components, local operations with global side effects can be described very naturally this way.

The meaning of a "statement" of an algorithm can now be defined as a mapping from states on states, where a state is a structure as in Figure 21. As an example consider the assignment statement. This statement might have the form "P.A := Q", where A is an accessor and PA and Q are paths within the local environment. (The statement should be read as "replace the A-component of $\overline{P}$ by $\overline{Q}$".) The meaning of the assignment statement could be defined as:

$$\begin{aligned}\mathcal{M}(P.A := Q) = {} & ADD(\overline{\Lambda}, p, \overline{loc.P})ADD(\overline{\Lambda}, q, \overline{loc.Q}) \\ & REM(\overline{p}, A)ADD(\overline{p}, A, \overline{q}) \\ & REM(\overline{\Lambda}, p)REM(\overline{\Lambda}, q).\end{aligned}$$

Notice that the following definition would not be correct:

$$\mathcal{M}(P.A := Q) = REM(\overline{loc.P}, A)ADD(\overline{loc.P}, A, \overline{loc.Q}).$$

The reason is that after $REM(\overline{loc.P}, A)$ both the object $\overline{loc.Q}$ and the path loc.P need no longer exist. The meaning of language constructs other than the assignment statement can be described in a similar way. For more details about this the reader is referred to [9].

## 4. CONCLUSION

In this paper a novel method of characterizing storage structures was discussed. The concept of a "structure" was introduced, which is basically a simple mathematical model of the access properties of a storage structure. Using this model storage structures with arbitrary sharing and circularities can be characterized without the need to introduce pointers. Creation and replacement become very natural operations which cannot produce any "garbage", since the concept of unreachability is nonexistent in a structure. Due to the fact that structures are general and yet free of such low level concepts as pointers and garbage, they lend themselves very well as the basis of definitions of realistic specification and programming languages. This is illustrated in [9], in which a specification language for abstract data types is discussed, which is used (in a somewhat informal way) in [10].

The concept of a structure as defined in this paper is believed to characterize storage structures in a way more abstract than other methods. In order to support this assertion let us give a short comparison of structures with some of these other methods. "Vienna objects" [14] are basically trees with labeled branches. Sharing and circularity can only be modeled by introducing a pointer concept. This is done by allowing "composite selectors" (which correspond to "paths") to be used as objects. "Graphs" [13] were already discussed in Section 2. Graphs are easily seen to be less abstract than structures, because each structure corresponds to many graphs. Also, the unnatural choice of an already existing node as the new node when creating a node in a graph is not necessary in a structure. "Relational objects" [5] are set-theoretic models of storage structures. They are built from atomic values using set and tuple constructors. Relational objects are more general than graphs (each graph can be described as a relational object), but they inherit many of the disadvantages of graphs. E.g., sharing can only be modeled by representing objects in some way as primitive values (which correspond to the nodes of a graph). The programming language SETL [4] even has a special atomic data type for this purpose. A more comprehensive comparison of structures with other methods of characterizing storage structures can be found in [9].

## REFERENCES

[1] BĒRZIŅŠ, V.A., Abstract Model Specifications for Data Abstractions, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (1979).

[2] BIRKHOFF, G., Lattice Theory, American Mathematical Society Colloquium Publications, Vol. XXV, American Mathematical Society, Providence, Rhode Island (1967).

[3] BIRKHOFF, G. & LIPSON, J.D., Heterogeneous Algebras, Journal of Combinatorial Theory 8, 115-133 (1970).

[4] DEWAR, R.B.K., The SETL Programming Language, To appear.

[5] EARLEY, J., Relational Level Data Structures for Programming Languages, Acta Informatica 2, 293-309 (1973).

[6] GOGUEN, J.A., THATCHER, J.W. & WAGNER, E.G., An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, In [15].

[7] GUTTAG, J.V. & HORNING, J.J., The Algebraic Specification of Abstract Data Types, Acta Informatica 10, 27-52 (1978).

[8] HOPCROFT, J.E. & ULLMAN, J.D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company, Reading, Massachusetts (1979).

[9] JONKERS, H.B.M., Abstract Storage Structures and the Specification of Abstract Data Types, To appear, Mathematical Centre, Amsterdam.

[10] JONKERS, H.B.M., Abstraction, Specification and Implementation Techniques in the Design and Classification of Algorithms, with an Application to Storage Management and Garbage Collection, Ph.D. Thesis, To appear, Mathematical Centre, Amsterdam.

[11] LISKOV, B. & ZILLES, S., Programming with Abstract Data Types, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9, No. 4, 50-59 (1974).

[12] LISKOV, B. & ZILLES, S., Specification Techniques for Data Abstractions, IEEE Transactions on Software Engineering SE-1, 7-19 (1975).

[13] MAJSTER, M.E., Extended Directed Graphs, a Formalism for Structured Data and Data Structures, Acta Informatica 8, 37-59 (1977).

[14] WEGNER, P., The Vienna Definition Language, Computing Surveys 4, 5-63 (1972).

[15] YEH, R. (Ed.), Current Trends in Programming Methodology, Vol. IV, Prentice Hall, Inc., Englewood Cliffs, New Yersey (1978).

[16] ZILLES, S., Procedural Encapsulation: A Linguistic Protection Technique, Proceedings of an ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, No. 9, 140-146 (1973).