# AN OUTLINE OF A MATHEMATICAL MODEL FOR THE DEFINITION AND MANIPULATION OF DATA

by

Bengt Nordström

Department of Computer Sciences
University of Göteborg and
Chalmers University of Technology
Göteborg, Sweden

Abstract

This paper presents a constructive model for data which suggests some precise definitions of notions like type, name, constants, declarations etc.
The model is based on an old idea of looking at data as a mapping from a set of names to a set of values. We will study two different kinds of assignments and also show how the model can be used to give some hints for new data structuring methods. In particular we have given a definition of a sequence and we have shown how this data structure can be used to give constructive definitions of structures like stacks, files and queues. We have also studied tree traversals i.e. the correspondence between various trees and sequences.

Key Words and Phrases: Data Type
Data structure description
High level languages
Set theoretic languages
Sequence
Tree traversal

CR Categories: 4.20, 5.24

## 0. Introduction

Recent articles (for instance 2, 3, 4) in the field of Computer Science have shown the necessity for strict definitions of some fundamental words connected to data. There is a vague intuitive notion of what words like data structure, data type, value, name and declaration mean. Some authors mean that a data structure is something like a matrix or a tree, that is something which has a structure when you denote it, as opposed to atomic items like integers, reals and so on. Some authors mean that a data type is a set of values, others mean that it is a set of values together with a set of operations defined on these values.

## 1. The relationship between a type and a data structure

From an abstract point of view the only difference between a finite subset of integers and a finite subset of trees are that they are sets with different kinds of functions defined from them. On the set of integers we usually define:

$$+: \text{int} \times \text{int} \rightarrow \text{int}$$
$$*: \text{int} \times \text{int} \rightarrow \text{int}$$
$$-: \text{int} \times \text{int} \rightarrow \text{int}$$

And on trees we usually define:

| | |
|---|---|
| left subtree: | tree → tree |
| right subtree: | tree → tree |
| depth: | tree → int |

and so on.

These functions are easier to visualize if we use different denotations for members of these sets and if we let the denotation reflect the characteristics of

the functions. The importance of denotation can for instance be seen in the influence of the poitional system for denoting integers and reals, and the introduction of the variable i in complex numbers.
It is therefore convenient to define a data structure to be a pair $<A,B>$ where A is a type (= a set of values) and B is a set of functions:

$$f: A^n \rightarrow C \text{ , } n = 1,2,\ldots \text{ where C is a type.}$$

This definition of a data structure is in accordance with the algebraic meaning of a (mathematical) structure.

## 2. A model for programming languages without references

In a computer program we manipulate values, which are in some sense a model of a part of the world outside the computer. We do this with the help of names, we let each name denote an object also called a value and we define functions from a subset of the values to another subset. Some names can change their values in the program, we will call these names variables. Other names cannot change their value after their initialization, we will call them constants. It has been shown convenient to restrict the set of values each name can denote. These subsets are called types.

Hence we will study three sets T (the set of types), V (variables) and C (constants). We will use greek letters to denote types.

We will call $0 =_{def} \bigcup_{\alpha \in T} \alpha$ the set of objects or values and the set $N =_{def} V \cup C$ the set of names.

We have two functions called declarations:

$$c: C \rightarrow T \quad \text{(constant declaration)}$$
$$v: V \rightarrow T \quad \text{(variable declaration)}$$

and val is a function

$$val: V \cup C \cup 0 \rightarrow 0$$

with the restrictions that

$$val(a) = a \text{ if } a \in 0$$
$$val(a) \in c(a) \text{ if } a \in C$$
$$val(a) \in v(a) \text{ if } a \in V$$

A computer program is a description how the function val is transformed, i.e. a program describes how the values which the names denotes will be changed during the execution of the program. We can look at a program as consisting of two parts: the declaration part and the algorithmic part.
The declaration part of a program usually consists of three parts:

### i. Type declaration part

This is a description of T, usually an enumeration of all its elements. This will also give a description of 0 and $val/_0$. (If f is a function $f: A \rightarrow B$ and if $C \subset A$ then $f/_C$ denotes the restriction of f to C.)
When we enumerate the types we can start from so called primitive types and construct new types from these. A primitive type is a type given in the language or a type defined by an enumeration of all its elements. For instance in PASCAL:

integer, real, color = (blue, red, yellow)

are all primitive types while

```
person = record  name: string,
                 male: bool end
```

is a new type constructed from earlier given types. We will discuss the construction of new types later in this paper.

### ii. Constant declaration part

This is an enumeration of all the elements in C, a complete description of the function c and usually

also a description of the function $val/_c$.

iii. Variable declaration part

This is similarly an enumeration of all the elements in V and a description of the function v.

In the algorithmic part of the program the function $val/_0$ will be defined and transformed.
The following simple example written in pseudo-PASCAL of a declaration part:

| | |
|---|---|
| color $=_{def}$ {blue, red, yellow}<br>int $=_{def}$ {1, 2, 3, 4} | type declarations |
| const a: color; const b, c: int<br>a = blue; b = c = 3 | constant declaration |
| var va: color; vb: int | variable declaration |

will be interpreted:

$T =_{def}$ {color, int} where color = {blue, red, yellow}
int = {1, 2, 3, 4}

$val/_0 =_{def}$ {(blue, blue), (red, red), (yellow, yellow), (1,1), (2,2), (3,3), (4,4)}

$C =_{def}$ {a,b,c}

$c =_{def}$ {(a, color), (b, int), (c, int)}

$val/_C =_{def}$ {(a, blue), (b, 3), (c, 3)}

$V =_{def}$ {va, vb}

$v =_{def}$ {(va, color), (vb, int)}

See also Figure 1.

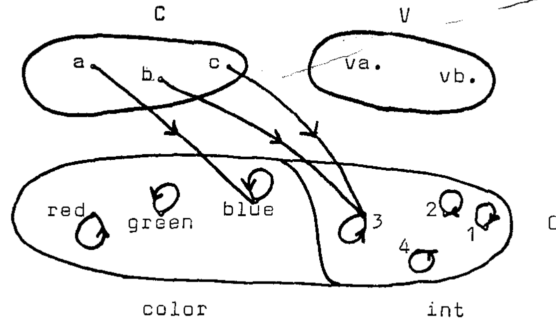



Fig 1: Example of the initialization of val in a program.

With this simple model we can describe programs in simple languages like FORTRAN, ALGOL etc. In order to allow for languages with references (pointers) we will iterate the model:

## 3. Extension of the model to references

For each set $\alpha \in T$ we define a set $\alpha_\uparrow$ (the set of all dynamically created references to $\alpha$) with a 1-1 mapping $f : \alpha_\uparrow \rightarrow \alpha$ and we extend val's domain to $\alpha_\uparrow$ by defining $val(b) = f(b)$ if $b \in \alpha$
We also define $0_\uparrow =_{def} \cup \alpha_\uparrow$ and for each $\alpha \in T$ we define two sets

$$\text{ref var}(\alpha) =_{def} \{b : v(b) = \alpha\}$$
$$\text{ref const}(\alpha) =_{def} \{b : c(b) = \alpha\} \cup \alpha_\uparrow$$

Let now $T_1 =_{def} \{\text{ref var}(\alpha) : \alpha \in T\} \cup \{\text{ref const}(\alpha) : \alpha \in T\}$

and $0_1 = \bigcup_{\alpha \in T_1:} \alpha = N \cup 0_\uparrow$

Now, we introduce two new sets $C_1$ and $V_1$ called constant references and variable references, and two functions called reference declarations:

$$rc: C_1 \rightarrow T_1$$

$$rv: V_1 \rightarrow T_1$$

We can now extend val's range to $0_1$ by:

$$\text{val}: C_1 \cup V_1 \rightarrow 0_1$$

with the additional restrictions that:

$$\text{val}(a) \in r\, c(a) \text{ if } a \in C_1$$
$$\text{val}(a) \in r\, v(a) \text{ if } a \in V_1$$

It is obvious how to change this last paragraph to be an induction step where we define $T_{n+1}$, $0_{n+1}$, $C_{n+1}$, and $V_{n+1}$, from $T_n$, $0_n$, $C_n$ and $V_n$.

The step from simple values to references when $T = \{\alpha, \beta\}$ is illustrated in Figure 2. For sake of simplicity we have assumed that no references are created dynamically.

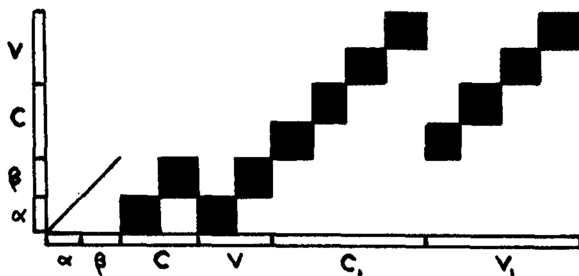



Fig 2: The range of val can be extended to names by introducing references. The diagram shows the possible values of val.

We can note that the model will allow for variable references to constants, variable references to variables, constant references to constants and constant references to variables. This is not the case in common programming languages, one exeption is MARY.

## 4. Assignments

In the algorithmic part of the program val is transformed by assignment statements, of which input and output are special cases. The semantics of such a statement varies from language to language. One axiom of assignment could be:

Cop Ass: $\{\text{val}(a_2) = w \wedge w \in \alpha \wedge v(a_1) = \alpha\}$
$a_1 := a_2 \quad \{\text{val}(a_1) = w\}$

We are using the notation of Hoare for expressing this axiom. It is important that an axiom of assignment is type invariant i.e. the restrictions of val will be left unchanged. It is easily proven that Cop Ass is type invariant. This axiom is called "assignment with copy" due to the fact that this assignment can only change the value of the name of the left hand side. This means that if two names denotes the same object one assignment can only change the value of one of them.

In many programming languages there is an automatic conversion between types if the types of the left hand side and the right hand side are not equal. This can be expressed formally as:

For some types $\alpha$, $\beta$ there is a function $C_{\alpha\beta}: \alpha \rightarrow \beta$ called conversion with the properties that

(i) $\quad C_{\gamma\delta} = C_{\alpha\delta} \circ C_{\gamma\alpha}$

if the right hand side exists.

(ii) $\quad C_{\gamma\gamma}(x) = x \quad$ for all $\gamma$.

The axiom of assignment with automatic conversion is now:

Conv Ass $\{\text{val}(C_{\text{Type}(a_2),\, v(a_1)}(a_2)) = w\}$
$a_1 := a_2 \quad \{\text{val}(a_1) = w\}$

$$\text{where Type}(a) = \begin{cases} v(a) \text{ if } a \in V \\ c(a) \text{ if } a \in C \end{cases}$$

In languages with references we have two particular important conversions:

$$C_{\text{ref } \alpha, \alpha} =_{\text{def}} \text{val}$$

and

$$C_{\alpha, \text{ ref const}(\alpha)} \text{ which is such that}$$

$$\text{val } (C_{\alpha, \text{ ref const}(\alpha)}(a)) = a$$

In ALGOL-68 terminology they are called dereferencing and referencing.

There are many more assignment statements. They are the object for further research. We are particularly interested to investigate what assignment statements are needed in languages without references.

## 5. Construction of new types

We have stated earlier that a type is nothing but a set of values. Hence the question of constructing new types from the primitive types is the same question as how to construct new sets from old ones in mathematics. We will therefore list some standard methods in set theory and show how these methods have their correspondence in programming languages - often disguised in the mantle of syntax.

### 5.1 Cartesian Product

One common way to create a new type is to define a cartesian product of two given sets. For instance the set of all complex numbers are:

$$\text{complex} =_{\text{def}} \text{real x real}$$

Usual names for cartesian products in programming languages are structured values (ALGOL-68), programmer defined data types (SNOBOL 4), record (PASCAL) etc. In most languages it is possible to give arbitrary names for the projections and it is common to use a special notation for the evaluation of these functions. For instance if we want to use the names im and re for the projections of the complex numbers we can do so:

a) struct complex = (real im, real re)    (ALGOL-68)

b) type complex = record im: real, re: real end    (PASCAL)

The notation for $P_1(a)$ if a is a name of type complex is then:

a) im of a
b) a. im

### 5.2 Functions

Given two sets A and B we can construct the set (A → B) of all functions from A to B. This is by far the most common way to construct new sets in programming languages. If f is a name of type (A → B) we will call f a procedure if f is a constant and a mapping if f is a variable. Procedures are commonly used in programming languages. In order to put them into our model we note that if f is a function of type (A → B) and if Type(a) = A then:

$$\text{Type}(f(a)) = B$$

and

$$\text{val}(f(a)) = \text{val}(f)(\text{val}(a))$$

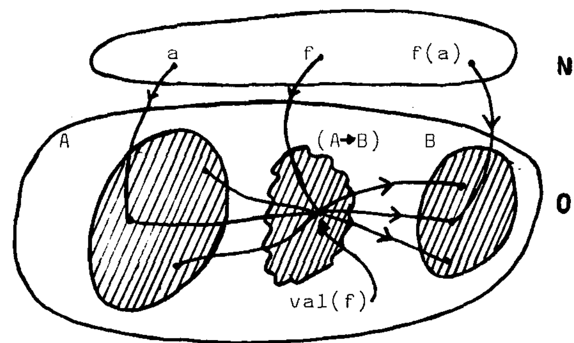



Fig 3: The relationship between val and a name of type function.

The use of mappings (i.e. the variable functions) has with two important exceptions been seriously neglected in commonly used programming languages. They offer a wide variety of implementation possibilities and need a further study. The two exceptions where the implementation is obvious are arrays and projections in cartesian products. An array is a mapping where the domain is an interval in a totally ordered set or a cartesian product of such sets. A set B is an interval if $a,c \in B \wedge a < b < c \Rightarrow b \in B$.
For instance an ordinary matrix can be declared:

$$\text{matrix} =_{\text{def}} [1..m] \times [1..n] \rightarrow \text{real}$$

## 5.3 Sequences

A sequence is an old familiar mathematical structure. In algebra it is common to define a sequence to be a mapping from the first n integers to a set. The set of all sequences of elements of A is then:

$$\sigma(A) = \{f: Z_k \rightarrow A \mid k \in Z_+\}$$

Obviously we can also define:

$$\sigma(A) = \bigcup_n A^n$$

where $Z_+$ is the set of all positive integers and $Z_k$ is the set of the first k integers.

Neither of these definitions is suitable for computer science applications like stack, file, buffers and so on. They will imply the existence of some unnecessary variables like an index for the projection operator. A formal description should be minimal with no extraneous details which would put unnecessary bounds to the implementation. We will therefore define a sequence from set theory which is suitable for specification of commonly used data structures.

### 5.3.1 Definition of a sequence

We will consider a set A to be well formed if for all $a \in A$

$$(\exists x)(x \in A)(x \in a) \Rightarrow (\exists! x \in A)(x \in a) \wedge (\forall z \in A, z \neq x)(z \neq \Phi \wedge A \cap z \subset \{\Phi\})$$

That means that if there is an element in a which is also an element of A then there is exactly one such element and the other elements of a are all different from $\Phi$ (the empty set) and their intersection with A contains at most the element $\Phi$. The most common case in programming languages is that if A is a type then $a \cap A = \Phi$ so all ordinary types are well formed. From a well formed set A we will construct the set $\sigma(A)$ in the following way:

(i) $\Phi \in \sigma(A)$

(ii) $\Phi \neq x \in A,\ y \in \sigma(A) \Rightarrow \{x,y\} \in \sigma(A)$

(iii) The set $\sigma(A)$ consists only of elements constructed by applying (i) and (ii) a finite number of times.

We will call an element of $\sigma(A)$ a <u>sequence</u> of A. This set $\sigma(A)$ has the following properties.

**$1^o$ $\sigma(A) \cap A \subset \{\Phi\}$**

Proof: Suppose that $z \in \sigma(A) \cap A$ and $z \neq \Phi$. Then $z = \{x,y\} \in A$, where $x \in A$ and $y \in \sigma(A)$. We can assume that $x \neq y$. (If $x = y$ then $x \neq \Phi$ and $x \in \sigma(A) \cap A$ and we can start from the beginning again. This process will stop after a finite number of times according to (iii).)
Then $y \cap A \subset \{\Phi\}$ which implies that $y = \Phi$ (because otherwise $y = \{s,t\}$ where $s \neq \Phi$ and $s \in A$). But this contradicts that A is well formed.

**$2^o$ All non-void members of $\sigma(A)$ has exactly two elements.**

Proof: Suppose that $\{x,y\} \in \sigma(A)$, where $x \in A$ and $y \in \sigma(A)$. If $x = y$ then $x = y = \Phi$ according to $1^o$. This contradicts (ii).

**$3^o$ $\sigma(A)$ is a well formed set**

Proof: Suppose that $a = \{x,y\} \in \sigma(A)$, $x \in \sigma(A)$, $y \in A$, $y \neq \Phi$. We notice that $2^o$ implies

that $y \notin \sigma(A)$. We want to show that $y \cap \sigma(A) \subset \{\Phi\}$. Suppose that $t \neq \Phi$, $t \in y \cap \sigma(A)$. Then $t = \{s, p\}$ where $s \in A$ and $s \neq \Phi$. Since A is well formed and $t \in y \in A$ and $t \notin A$ we know that $A \cap t \subset \{\Phi\}$ which contradicts that $s \neq \Phi$.

From $3^o$ we can conclude that the set $\sigma(\sigma(A))$ can be formed.

If $x \in \sigma(A)$, $x \neq \Phi$ we will call the two elements of x first (x) and rest (x), where first $(x) \in \sigma(A)$.

Definition:

$1^o$ rank: $\sigma(A) \to N$

rank(s) $=_{def}$ (s=$\Phi$ | 0 | rank(rest(s)) + 1)

$2^o$ if $x, y \in \sigma(A)$ we define:

$$\begin{cases} \Phi + y = y \\ x + y =_{def} \{\text{first}(x),\ \text{rest}(x) + y\} \end{cases}$$

$3^o$ $x \leq y$ (x is a final sequence of y) if $x = y$ or $x \leq$ rest(y)

We have used the ALGOL-68 if-clause in the first definition. The sum in the second definition is well defined according to a proof or induction over the rank of x (rank (rest(x)) = rank(x) -1).

We can now list some more properties of $\sigma(A)$. We assume $x \in \sigma(A)$ .

$4^o$ $<\sigma(A), + >$ is a monoid.

Because $\Phi$ is a zero-element since $y + \Phi = \{\text{first}(y),\ \text{rest}(y) + \Phi\}$ = (induction) = $\{\text{first}(y),\ \text{rest}(y)\} = y$

and it is associative since

$(x + y) + z = \{\text{first}(x),\ \text{rest}(x) + y\} + z = \{\text{first}(x),\ (\text{rest}(x) + y) + z\}$ = (induction) = $\{\text{first}(x),\ \text{rest}(x) + (y + z)\} = x + (y + z)$

$5^o$ There is an isomorphism $C_n$ between $A^n$ and $\{x \in \sigma(A) : \text{rank}(x) = n\}$ this is obvious: if $P_i$ is the projection of the cartesian product then $C_n$ is such that $P_i\ (C_n^{-1}(x))$ = first $(\text{rest}^{i-1}(x))$, $1 \leq i \leq n$

(if $f : X \to X$ then we define

$f^o: X \to X : x \mapsto x$

$f^n: X \to X : x \mapsto f(f^{n-1}(x))$)

This isomorphism gives us a hint to another denotation for a sequence:

if $x \in \sigma(A)$, rank(x) = n and $C_n^{-1}(x) = (a_1, a_2, \ldots, a_n)$ then we will write $x = <a_1, a_2, a_3, \ldots, a_n>$

We can now extend the definition of addition:

Definition:

if $x \in \sigma(A)$, $y \in A$ then

$$\begin{cases} x + y =_{def} x + C_1\ (y) \\ y + x =_{def} C_1\ (x) + y \end{cases}$$

and so if $x \neq \Phi$ we have

$6^o$ x = first(x) + rest(x)

5.3.2 Definition of some well known data structures

We can now use this definition of a sequence to define the semantics of some familiar data structures. In defining these structures we will use a mixture of ALGOL-68 and PASCAL with some minor exceptions. We will write a cartesian product

complex $=_{def}$ (im: real, re: real)

or complex $=_{def}$ (im, re: real)

The syntactic unit a type will be substituted for a given type at the declaration of a variable. Instead of writing '$\sigma(A)$' we will write 'sequence of A' and instead of '$\Phi$' we will write 'empty' .

a) STACK of a type $=_{def}$ sequence of a type

proc top (x: stack of a type): a type $=_{def}$
if x ≠ empty then top : =
= first(x); x : = rest(x) fi

```
proc push (x: stack of a type, y : a type) :
      stack of a type=def
      push: = y + x

proc pop (x : stack of a type):
      stack of a type = def
      if x ≠ empty then pop : = rest(x) fi
```

```
b) FILE of a type =def(left,right: sequence of a type)

   proc open (x:file of a type) =def
         x: = (empty, empty)

   proc put (x: file of a type, y:a type) =def
         with x do if right = empty then
                  left:= left + y fi od

   proc reset (x : file of a type) =def
         with x do right: = left; left: = empty od

   proc get (x: file of a type): a type =def
         with x do if right ≠ empty then
                  get:=    first (right);
                  right: = rest (right);
                  left:=   left + get
                  fi
               od
```

```
c) FIFO of a type =def sequence of a type

   proc put (x: fifo of a type, y: a type) =def
         x: = x + y

   proc get (x: fifo of a type): a type =def
         if x ≠ empty then get: = first (x);
                           x: = rest(x)
                  fi
```

These definitions are all constructive in the sense that they are easily implemented on a computer. At the same time they are sufficiently abstract to allow for different implementations. P. Lucas has defined that y is a stack if:

$$top(push\ (y,x)) = x$$
$$pop(push(y,x)) = y$$
$$y \neq empty \Rightarrow push(pop(y), top(y)) = y$$

are all true. It is easy to prove that the above definition of a stack has these properties.

### 5.3.3 Traversal of trees and forests

Familiar data structures in programming languages are binary trees and forests. We can define them recursively in the following way, we assume for simplicity that the nodes are integers:

binary tree $=_{def}$(info:int, left,right:binary tree) ∪ {empty}

forest $=_{def}$sequence of (info:int, subforest: forest)

In figure 4 we give some examples of values of these types.

(1, Φ, Φ)

(3, (2, Φ, Φ), (1, Φ, Φ))

(4, (3, (2, Φ, Φ), (1, Φ, Φ)), (1, Φ, Φ))

<(2, Φ), (3, Φ)>

<(1, <(2, Φ), (3, Φ), (4, Φ)>) ,
(5, <(6, Φ), (7, <(8, Φ), (9, Φ)>)>)>

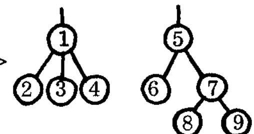



Figure 4: Examples of binary trees and forests

When we are working with trees or similar data structures we often want to examine the infos of the structure in a certain order. We can make the following

Definition: A traversal is a mapping from a binary tree or a forest to a sequence.

The definition can be generalized to other data structures. The three most important ways to traverse a binary tree are preorder, inorder and postorder. We can for instance define:

```
proc preorder(t:binary tree): sequence of int =def
    if t = empty then preorder := empty
                 else with t do preorder := info +
                                preorder (left) +
                                preorder (right)
                            od
                 fi
```

We get the other standard traversals by permuting the terms in the sum in the procedure.

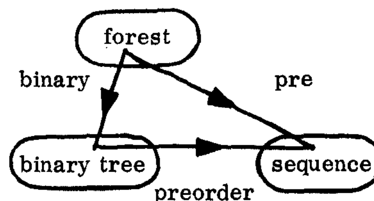
It is of course possible to define a similar traversal for a forest:

```
proc pre (f:forest): sequence of int =def
   if f = empty then pre := empty
                else pre := first(f).info +
                     pre(first(f).subforest) +
                     pre(rest(f))
                fi
```

There is one important transformation between a forest and a binary tree. It is called the natural correspondence and is defined by:

```
proc binary(f:forest):binary tree =def
   if f = empty then binary := empty
                else binary := (first(f).info,
                                binary(first(f).subforest),
                                binary(rest(f))
                fi
```

We thus have three functions defined between forests, binary trees and sequences:




The following theorem should not be a surprise:

Theorem: The diagram above commutes (i.e. pre = = preorder o binary)

Proof: If $f \neq \Phi$ is a forest then

binary(f) =(first(f).info, binary(first(f).subforest), binary(rest(f)))

preorder(binary(f))= first(f).info+

+ preorder(binary(first(f).subforest))

+ preorder(binary(rest(f)))

An induction over number of infos will yield the result.

The definition of a traversal should be reflected in the control structure of a language. This means that if there is a for-clause:

for x in s do ... od

which makes x run through all the elements of the sequence s we should also have the possibility to write:

for x in f(t) do ... od

where f is a traversal. This then makes x traverse t in the order defined by f.

The advantage of this is that we can use the same language construct (the for-clause for a sequence) to traverse different data structures. We can for instance write:

```
for x in preorder(t) do
for x in anyorder(s) do
for x in rowwise(m) do
```

where t is a tree, s is a set and m is a matrix.

In order to illustrate the usage of sequence and trees we will present two examples picked from Knuth (10). The first example is a radix list sort. It is a sorting method used in mechanical card sorters, based on the digits of the keys. We distribute the cards according to the value of the least significant digit into different piles. Then we collect these piles and make another distribution based on the next digit, and so on. See Knuth for a more detailed explanation. In the procedure below we assume that the keys are p-tuples

$$(a_p, \ldots, a_2, a_1) \qquad 0 \leq a_i \leq M$$

and the order is defined lexicographically.

```
proc radix_list_sort (file: sequence of keys)
begin pile: array ( 0..M) of sequence of keys;
    for k:= 1 to P
        do
        comment distribute the keys;
        for S in file
            do
            i:= digit_nr (k): of (S);
            pile (i):= pile (i) + S
            od;
        comment collect the piles;
        file:= empty;
        for l:= 0 to M
            do
            file:= file + pile (l);
            pile(l):= empty;
            od
        od
end
```

The second example is an algorithm for finding a tree with minimum weighted path length. The algorithm works as follows: We assume that we are going to construct a tree (= a forest with one element) based on the weights $\langle w_1, w_2, \ldots, w_m \rangle$ (these are in increasing order). We solve the problem for m - 1 weights $\langle w_1 + w_2, w_2, \ldots, w_m \rangle$ and replace the element $\boxed{w_1 + w_2}$ in this solution by

```
proc huffman (s : forest);
comment s is a forest of depth 1 and
        contains the weights;
begin
nr 1, nr 2, t: (info:int, subforest:forest)
while rest(s) ≠ empty
    do
    nr 1:= first(s); nr 2:=first(rest(s)); s:=rest(rest(s));
    t:=(nr1.info + nr2.info, <nr1, nr2>);
    for ss ≤ s
        do comment insert t into its proper place;
        if first(s).info ≥ t.info then ss:= t+ ss; exit
                                      fi
        od
    od
s:= t; comment s is now transformed to a tree with
               minimum weighted path length;
end
```

This example uses the syntactic construction 'for ss ≤ s do..od' which makes ss be the name of all final sequences of s in decreasing rank, i.e. it will be equivalent to:

ss:= s; while ss ≠ empty do...ss:= rest(ss) od
except that ss will be locally declared within do..od and all assignments to ss will also affect s.

### 5.4 Unions

Another way to create a set is to take the union of two sets. In order to gain security it is often necessary to check from which set in the union a certain element comes. This can be accomplished by a set sum between the two sets.

A set sum is defined by:

$$\alpha + \beta =_{\text{def}} \{(a,1): a \in \alpha\} \cup \{(b,2): b \in \beta\}$$

In order to see whether an element comes from $\alpha$ or $\beta$ we only check if there is a 1 or a 2 stuck to the element. The obvious conversion from a set sum to one of its terms is called deuniting in ALGOL-68.

### 5.5 Other ways

Other ways to create new types include: taking a subset of a given set with a condition for the members of the set. We can also create the powerset of a given set i.e. the set of all subsets of a given set. This has been incorporated in PASCAL in order to formalize bit operation on words in computers. It should be possible to use other operations on powersets than union, intersection and membership tests. It should also be possible to define functions and operations on these subsets. Another way to say this is to ask for the possibility of variable types (variable in the sense that the number of elements can vary), where we can make set operations on a type dynamically in a program. Such a set is easily implemented, for instance as a cartesian product of a bitstring and an array.

## 6. Conclusions

A basic assumption behind the ideas of data structures presented in this paper is that we can use set theory in describing the semantics of data structures. Set theory is familiar to most of us and almost all concepts of set theory have their counterpart in data types. We have given some hints where it could be worthwhile to incorporate more concepts of set theory in programming languages.

Acknowledgements. I have benefited in this work from discussions with Åke Wikström.

## References:

1. G.H. Mealy:
"Another look at Data" in FJCC 1967 pp 525-534

2. R.R. Korfhage:
"On the Development of Data Structures"
SIGPLAN Notices, Dec 1974

3. A.N. Haberman:
"Critical Comments on the Programming Language PASCAL"
Acta Informatica, 47 - 58 (1973)

4. Lecarme/Desjardins:
"Reply to a Paper by A.N. Haberman on the Programming Language PASCAL"
SIGPLAN Notices 9, 21 - 27 (oct 1974)

5. N. Wirth:
"The Programming Language PASCAL"
Acta Informatica 1, 1 pp 35 - 63

6. Conradi/Holager:
"A Study of Mary's Data Types in a Systems Programming Application" IFIP/TC2 Working Conference on Machine Oriented High Level Languages pp 295 - 309

7. ed. van Wijngaarden:
"Report on the Algorithmic Language ALGOL 68"
Num. Math. 14 pp 79 - 218

8. C.A.R. Hoare:
"An Axiomatic Base for Computer Programming"
CACM 12 pp 576 - 580 (1967)

9. P. Lucas:
"On the Semantics of Programming Languages and Software Devices"
in Formal Semantics of Programming Languages
ed. Rustin pp 41 - 48

10. D. Knuth:
"The Art of Computer Programming"
Addison - Wesley

11. Maclane/Birkhoff:
"Algebra"
The Macmillan Company