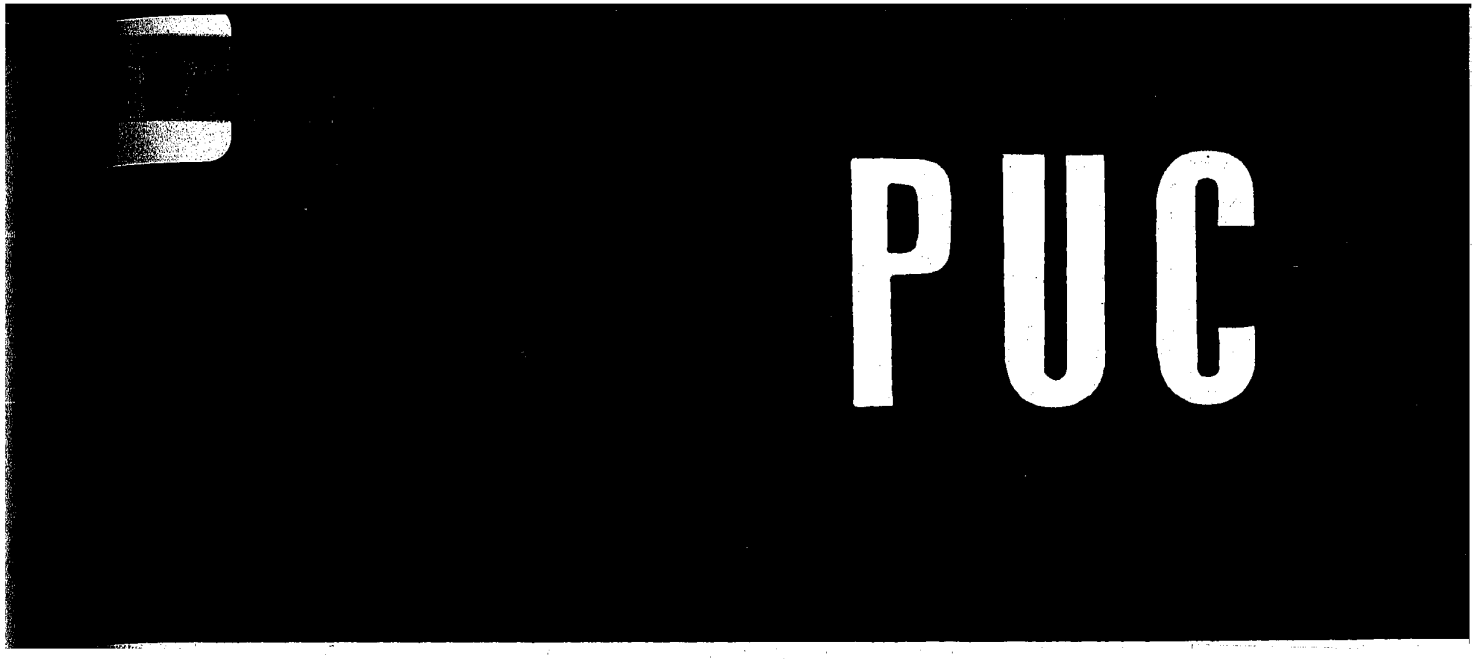


Series: Monographs in Computer Science
and Computer Applications
Nº 4/75

ISSUES IN DATA TYPE CONSTRUCTION FACILITIES

by

Carlos J. Lucena
Daniel Schwabe
Daniel Berry

Computer Science Department - Rio Datacenter

Pontificia Universidade Catolica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 4/75

ISSUES IN DATA TYPE CONSTRUCTION FACILITIES *

by

Carlos J. Lucena
Associate Professor
Computer Science Department
(Dep. Informática - PUC/RJ)

Daniel Schwabe
Research Assistant
Computer Science Department
(Dep. Informática - PUC/RJ)

Daniel Berry
Assistant Professor
Computer Science Department
University of California, Los Angeles

Series Editor: Larry Kerchberg

August, 1975

* This work was supported in part by the Brazilian Government Agency FINEP under contract Nº 244/CT and the NSF/CNPq program in Computer Science

ABSTRACT:

This work deals with the problem of synthesizing correct data representations for data abstractions. An extension of the "cluster" approach is proposed that also addresses the problems of efficiency and portability.

KEY WORDS:

Cluster, data representation, data abstraction, portability, very high level languages, set theoretic types.

RESUMO:

Este trabalho trata do problema de síntese de representações de dados corretos para abstrações de dados. Propõe-se uma extensão da técnica de "clusters" que considera adicionalmente os problemas de eficiência e portabilidade.

PALAVRAS CHAVE:

"Cluster", representações de dados, abstrações de dados, portabilidade, linguagens de nível muito alto, tipos que são conjuntos.

1. INTRODUCTION

More and more, computer scientists everywhere are trying to deal with the problem of the high cost of software. One of the key reasons for this high cost is the gap between program specification and program implementation. One approach to closing this gap has been to develop means to provide language facilities for at least informally proving consistency between the specification and the implementation of the program [HOA72,LIZ75].

In this work, we explore a variation of one existing approach of bridging the gap, namely the cluster approach. The variation that we follow is aimed at synthesizing correct data representations for data abstractions we propose an extension of the cluster approach that will also address the problems of efficiency and portability.

The cluster approach is due to Liskov and Zilles [LIZ74] and consists of some language features to model and implement abstract types in terms of operations applicable to objects of the type in such a way that the user of the type needs to be concerned only with the abstract behavior of the type as presented by the operations.

A cluster is an independent external module in which the initialization of the representation of an abstract type T is performed, and in which all operations P_j $(1 \leq j \leq n)$ related to T are performed. The heading of a cluster has the following form:

T : cluster <cluster formal parameters list option>

is $P_1, P_2, \ldots, P_n$

The body of the cluster consists of the declaration of the representation (which is visible throughout the cluster) and the code for each P_j.

For more details on clusters, consult [LIZ74,LIS74]

The language features that we will be discussing in this paper are extracted from the programming language PEP (a language for Provability, Efficiency, and Portability) [CAR75]

which was designed at PUC(*). In the PEP language, the specification level consists of a very high level language (VHLL) of the SETL-MADCAP family [MOR72,MLM74,SCH71,SCH72,MUG73].

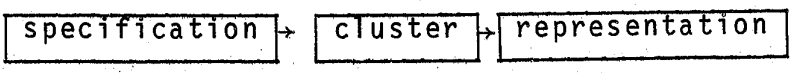
Clusters are used at the specification level of PEP to introduce abstract data types that can be used in addition to the standard set theoretic types provided in PEP [CAR75].

Cluster are also used to re-state the semantics of a possibly general standard type in terms of more basic standard types in the language. In this case, the code produced by the compiler for the data representation and the related operations defined by the cluster will be used in lieu of the code normally produced for the replaced very high level code. In both situations we are interested in the efficiency of the code generated from the clusters.

Another design objective we had for PEP was to be able to compile its programs into efficient code for a wide variety of base machines. To our advantage, we found that the portability question could be answered by the same solutions used to answer the efficiency question. Finally, our proposed revisions of the original concept of cluster enchances its ability to support stepwise construction and proof of implementations of abstract data types.

2. THE PEP METHODOLOGY

As we stated in the introduction, one of the objectives of PEP is to reduce the gap between the specification of an algorithm for the solution of some problem and its coding into a machine acceptable form. For this and other reasons, PEP organizes a program into three distinct levels.



The topmost level is called the specification level. In this level the programmer specifies his algorithm using a

---

(*) Pontifícia Universidade Católica, Rio de Janeiro - Brasil

VHLL. At this level he has available to him, the usual scalar data types such as integer, real, etc., as well as the following structured data types:

array as in Algol 60 [NAU63]
record as in PL/1 [IBM72]
set (of homogeneous elements)as in MADCAP or SETL [MOR72,MUG73]
sequence (of homogeneous elements) as in MADCAP or SETL

Furthermore, through the use of our modified cluster concept the programmer may use abstract types defined in clusters at the second level.

The second level is the cluster level. In a PEP cluster, the programmer may define the abstract type in terms of a flexible representation and operation bodies written using a standard instruction set for the flexible representation. The standard operations include create, add, sub, select, etc., for creating instances of the data, adding components, subtracting components, etc...

The behavior of the standard instructions on the flexible representation is specified by axioms. The flexible representation will usually not be a fixed concrete representation available on the base machine, but rather a higher level representation implementable in several ways.

The actual implementation of the flexible representation is specified in the bottommost level, called the representation level. This level is considered the machine level. Here, the programmer selects a fixed (concrete) representation satisfying the axioms of the flexible representation. Usually (hopefully), these fixed representations will be available from a library; however, they may be coded in yet another cluster. At this level, the standard instruction set is implemented in terms of machine level operations as efficiently as possible.

The three levels supported by PEP may be compared with the three levels of abstraction suggested by Earley [EAR73].

The data types we use in PEP'S specification level correspond to Earley's relational level (data are described in terms of relations between the data items), our cluster level corresponds to Earley's access path level, and our representation level corresponds to Earley's implementation level.

The use of flexible representations in clusters plus the possibility of having libraries of fixed representations for each base machine will allow us to achieve portability. Performance evaluation of the various commands in the standard instruction set for a set of concrete representations in a particular library will allow us to select a minimum cost fixed representation for a given application [TOM75,GOT74,LOW74] thus allowing us to achieve the goal of efficiency. Let us consider an example showing two of the levels in PEP.

3. EXAMPLE

Before presenting the example, it is necessary that we specify the semantics of the PEP *if* command; the syntax is

```
if
    B1  then  S1;
    B2  then  S2;
    ..............
    Bn  then  Sn;
i-end
```

$B_1,\ldots, B_n$ are boolean expressions and $S_1,\ldots, S_n$ are statements. The boolean expressions are evaluated in order of appearance, and the statement S_i following the first true B_i is executed followed by the instruction after *i-end*. If none of the boolean expressions are true, the instructions following *i-end* are executed next. For more detailed information see [CAR75].

Figure 1 contains a specification level PEP program to compute the minimum cost spanning tree of a graph as specified in example 4.1 on page 109 of Aho, Hopcroft and Ullman (AHU75). The algorithm given here is a direct translation of figure 4.1 on pag. 110.

```
1   minimum-cost-spanning-tree: proc (reads n: integer);
2   type arcs: set of edges; type vertices: set of vertex;
3   type edge: tuple of vertex;
4   type vertex: subrange (1..n);
5   type spanning: set of (set of vertex) ;
6   id E,T: arcs;
7   id VS←{}  : spanning;¢ initializing declaration ¢
8   id W1, W2, V: vertices;
9   id v,w : vertex;
10  read  E and V;
11  T←{} ;
12  for v in V do VS← VS ∪ {V};
13  while card(VS)>1 do
14     pick one <v,w> in E such that
       (V<s,t> in E) (cost (<s,t)≥ cost(<v,w>));
15     E←E - {<v,w>};
16     W1← unique s in VS such that v in s;
17     W2← unique s in VS such that w in s;
18     if W1 ≠ W2 then begin
19        replace W1 and W2 by W1 ∪ W2
20        T←T ∪ {< v,w >}
21       end
22     i-end
23  w-end
24  p-end minimum-cost-spanning-tree
```

Figure 1

Three sets are used: E which contains the edges of the graph, V which contains the vertices of the graph and T which is used to collect the edges of the minimum cost spanning tree. The algorithm transforms a spanning forest into a single tree according to the cost function. The set contains the trees in the spanning forest.

Initially, the forest contains one tree for each vertex consisting of only that vertex. Each cycle through the loop expands one tree to include the vertices of another and reduces the size of the forest VS by one.

Line 11 sets T to the empty set of edges. Then, line 12 initializes VS to contain each vertex of G in a singleton set. Line 13 sets up a loop which cycles until the size of VS's down to one set containing all vertices of G. In the body in line 14, an edge <v,w> of minimum cost is selected and then in line 15 removed from E. Lines 16 and 19 find the unique sets W1 and W2 which contain v and w. If these two sets are different then in line 19, they are replaced by their union.

Finally the edge <v,w> is added to T in line 20. When the loop is done all vertices are in the one set VS and T contains the edges of a minimum cost spanning tree.

If we examine the program (as did, Aho, Hopcroft, and Ullman) we notice that each vertex is in exactly one set in VS and thus the sets in VS are disjoint. Consequently:

a) The sets in VS could be named by integers; an equality check on the names implements the set equality check.

b) Actually, we do not really need to know how the sets are named, for there is no explicit mention of any set name anywhere in the program.

c) Since the sets in VS are disjoint, the replace--by-union in line 13 could be implemented more efficiently than the standard set union operation which has to check for repetitions.

d) The operations to find the unique sets containing specific elements in lines 16 and 17 could return a set name.

```
 0  minimum cost-spanning-tree: proc (reads n: integer);
 1  type arcs: set of edges; type  vertices: set of vertex;
 2  type edges: tuple of vertex;
 3  type vertex: subrange (1..n);
 4  type spanning: set of (set of vertex) whith cluster (n);
 5  id n integer;
 6  id E,T: arcs;
 7  id  VS: spanning;¢ initializes VS to {}¢
 8  id W1, W2, V: vertices;
 9  id v,w: vertex;
10  read E,V, and n;
11  T←{};
12  for v in V do VS$add(v);
13  while VS$# > 1 do:
14       pick one <v,w> in E such that
         (∀<s,t> in E)(cost(<s,t>)≥ cost(<s,w>));
15       E←E - {<v,w>};
16       W1← VS$find(v);
17       W2← VSfind(w);
18       if W1≠W2 then begin
19            VS$replace-by-union(W1,W2);
20            T←T  ∪ {<v,w>}
21            end
22       i-end
23   w-end
24   p-end minimum-cost-spanning-tree;
```

Figure 2a

```
1     spanning: cluster (n: integer) on rep2(repl (integer))
      is replace-by-union, find, add, # ;
2     id size: integer array [1:n];
3     id VS: rep;
4     id number-of-sets: integer;
5     id name-generator: integer;
6     create
7     number-of-sets ← 0;
8     name-generator ← 0;
9,10  for n times do VS<0>$add('-',0) f-end
11    endcreate;
12    replace-by-union: op (a,b): integer);
13    id k, element: integer;
14    if size (a) >size (b) then a<->b i-end ¢ see footnote ¢
15    element ← VS<a ∘1>;
16    K ← 1;
17    while element ≠ Λ do
18       VSreplace (0∘element, b);
19       k +←1; ¢ see footnote ¢
20       element ← VS<a ∘1> ;
21    w-end;
22    VS<b>$add(1,'-'<a>);
23    VS$replace(a,Λ);
24    size (b) ← size(a) + size (b);
25    number-of-sets-←1;
26    end replace-by-union;

27    find: op (i: integer) returns (integer);
28    returns(VS<0 ∘i>);
29    end find;
```

Figure 2b

"k+←1" means "k←k+1". Likewise for -← etc;
"a←→b" means "exchange values of a and b".

Figure 2b cont'd

```
30   add: op(e: integer);
31        name-generator +←1;
32        number-of-sets +←1;
33        size(name-generator)←1;
34        VS$add('-',Λ);
35        VS<name-generator>$add(a)
36        VS$replace(0 ∘S, name-generator);
37   end add;
38   # : op() returns (integer);
39       return (number-of-sets);
40   end # ;
```

It was therefore decided to use a cluster to implement the type spanning in a more efficient manner. Figure 2a gives the program with references to spanning and operations on VS changed to access the cluster, and figure 2b gives the cluster itself.

The cluster implements a set of sets; therefore the flexible representation (rep) used has two levels: the outer level is rep2 and the inner is rep1 Thus, rep1 models sets that contain integers and rep2 models sets containing sets of integers. Notice that we do not specify what concrete data structure each of rep2 and rep1 is. They could be interpreted as an array (rep2) of lists (rep1), a list (rep2) of lists (rep1), or anything else the programmer desires. The operations defined in the cluster are create, replace-by-union, find, add, and # .

The two level rep used is inspired by the data structure used in [AHU74]. Element i of VS (of type rep) contains the elements of the set named "i". Element 0 of VS contains a mapping from an integer vertex to the name of the set containing the vertex; in position i of VS<0> is the name of the set containing i. This mapping is used to optimize the find operation.

The array size contains the size of each set, number-of-sets contains the number of different sets in VS, and

name-generator is used to assign a unique name to a new set being added to the set of sets VS.

The keywords create and endcreate enclose a sequence of statements that initialize VS to an empty set of sets. These statements are executed each time a variable of type spanning is declared in the specification level program.

The operations < >(which is "syntactic sugar" for" select"), replace and add used in lines 10,15,19,22, etc... are all in the standard instruction set of rep1 and rep2 with identical semantics in each case.

For each element in the smallest of the two sets a and b, replace-by-union changes the name of the set containing the element as given by VS<0> (lines 14-2) and then adds these elements to the end of the larger set (line 22). The smaller set is then wiped out (line 23), thus reducing the number of sets in VS by one (line 25). The symbol ∘ denotes composite selection, that is, VS<a> selects the entire a th element of VS and VS<a ∘k> selects the kth element of the ath element of VS.

Because of the way the rep is organized, to find the name of the set i we have only to look up VS<0 ∘i>; this is precisely what the body of find does (line 28). To add a new set containing the one element e to the set of sets VS, we have to give the new set a new unique name, increase the number of sets, set the size of the new set to one, add an empty set to the end of VS, update that set to a singleton set containing e, and finally update VS<0> so that the setname of e is name-generator.

The body of # has only to consult the variable number-of-sets (line 39). This variable has been kept up to date by the other operations.

We have shown the process of replacing a high level by a cluster for only one type. This process may be done for all of the high level in the program. In the next section, we consider the selection of a fixed representation for the flexible representation, rep, of the cluster.

4. PORTABILITY

Our approach to portability is based on the idea of standardizing the instruction set that operates on the flexible representation. This notion was suggested by the work of Standish [STA73] where an attempt is made to axiomatize the basic properties of all data structures.

Assuming a storage structure that behaves like Standish's data spaces we have converted the selection and assignment operations into a set of convenient basic operations that form the instruction repertoire of our cluster level (i.e. of our flexible representations). These basic operations heve a very simple and universal semantics which does not depend on the actual implementation of the flexible representation.

Let $r \in rep(t)$. Then a value of r is a sequence of t's selected by consecutive integers starting from 1. The length of such a sequence is one less than the first index j such that the selection of the j^{th} element yields Λ. The operations assumed are

add: r x{'+','-'}x t → r
add: (S,p,e)$\triangleq$ add e to the beginning or end (depending on whether p is'+' or '-')of S
sub: r x t → r
sub: (S,e)$\triangleq$ remove e from S if e is in S
select: r x int → t
select: (S,i)= S<i> $\triangleq$ the i^{th} element of S if it exists and Λ otherwise
replace: r x int xt → r
replace: (S,i,e)$\triangleq$ change the i^{th} element of S to e if S<i>$\neq\Lambda$
insert: r x int x {'+','-'} x t →r
insert: (S,i,p,e)$\triangleq$ insert e into S before or after (depending on whether p is '+' or'-') the i^{th} element of S.

On the above whenever an element is inserted or removed the indices are shifted to preserve the fact that consecutive integers from 1 through the length select non-Λ elements. Formally, we have the following axioms:

Let r = rep (t)

1) $v \in r \supset [\forall i)(1 \leq i \leq \text{length } (v) \supset (\text{select}(v,i) \in t \wedge \text{select}(v,i) \neq \Lambda)]$

2) $v = \text{add}(s,'+',e)$ iff $[\text{length } (v) = \text{length}(s)+1 \wedge \text{select } (v,1) = e \wedge ((\forall i)\ (1 \leq i \leq \text{length } (s) \supset \text{select } (v,i+1) = \text{select } (s,i)]$

3) $v = \text{add } (s,'-',e)$ iff $[\text{length } (v) = \text{length } (s)+1 \wedge \text{select } (\text{length } (v)) = e \wedge ((\forall i)\ (1 \leq i \leq \text{length } (s) \supset \text{select } (v,i) = \text{select } (s,i)))]$

4) $v = \text{sub } (s,e)$ iff
$(\exists j \ni e = \text{select } (s,i) \supset [\text{length } (v) = \text{length } (s-1) \wedge (\forall i)\ (1 \leq i \leq j-1 \supset \text{select } (v,i) = \text{select}(s,i) \wedge ((\forall i)\ (j+1 \leq i \leq \text{length } (s) \supset \text{select } (v,i-1) = \text{select}(s,i)))] \wedge ((\exists j) \ni e = \text{select } (s,j)) \supset v = s$

5) $v = s$ iff $(\forall j)(\text{select } (v,j) = \text{select } (s,j))$

6) $v = \text{replace } (s,i,e)$ iff
$((\text{select } (s,i) \neq \Lambda) \wedge (j \neq i \supset \text{select } (v,j) = \text{select } (s,j)) \wedge \text{select } (v,i) = e)$

7) $v = \text{insert}(s,i,'+',e) \supset [\text{length } (v) = \text{length}(s)+1 \wedge ((\forall j)(1 \leq j \leq i) \supset \text{select}(v,j) = \text{select } (s,j)) \wedge ((\forall j)\ (i+1 \leq j \leq \text{length}(s) \supset \text{select } (v,j+1) = \text{select } (s,j)) \wedge \text{select } (v,i+1) = e]$

8) $v = \text{insert}(s,i,'-',e) \supset [\text{length } (v) = \text{length } (s)+1 \wedge ((\forall j)(1 \leq j \leq 1-1) \supset \text{select}(v,j) = \text{select}(s,j)) \wedge ((\forall j)(i \leq j \leq \text{length}(s) \supset \text{select}(v,j+1) = \text{select}(s,j) \wedge \text{select}(v,i) = e]$

Notation: $v\langle i \rangle = \text{select}(v,i)$

For the objective of portability to be achieved, each machine for which we want to compile a PEP program has to host a library of fixed representations in its own machine language.

Furthermore, this library has to include implementations of every one of PEP's standard types. In other words, what we really have made portable are the source language at the specification and cluster levels and PEP's control structures.

Thus, for example, we could, as already indicated, move to any machine and select any available fixed representations for our two flexible representations rep2 and rep1.

In actuality not just any fixed representations will be selected, but rather, representations that are believed to be more efficient will be chosen. The analysis in [AHU74] suggests that it would be best to choose array as the implementation of rep2 and linked-list with updating in place as the implementation of rep1. Thus, we can achieve any desired efficiency.

Tompa and Low [TOM75,LOW74] have suggested a methodology for selecting efficient implementations of data types which fits well with our proposal. Each fixed representation should have a cost table permitting the computation of a weighted cost function.

The table gives for each standard operation the cost of using the operation (in space and/or time). Given this table plus an expected or actual frequency distribution of calls to the operations for a particular problem, it is possible to weight the cost of each operation to obtain a total cost measure for the fixed representation applied to the problem. The total cost measure of several fixed representations for a problem may be compared to select the most efficient fixed representation. It is not difficult to envisage a system which automatically gathers use statistics on calls to standard operations, automatically computes the total cost measure for the available fixed representations and automatically selects the most efficient fixed representation.

Thus, the claim we make in this paper is that methodology and features of PEP do support portability and efficiency.

5. PROOF ASPECTS

It turns out that the methodology supported by PEP's features also help in discovering proofs of the correctness of a data type implementation. Normally, one implements an abstract type by giving a cluster with a fixed representation with fixed operations. In proving this cluster correct, one must obtain a direct mapping from the fixed representation to the abstraction [HOA72]. The difficulties are:

1) finding the mapping
2) if the representation is changed, the proof must be totally redone right down to the mapping.

The proposed three level type construction methodology helps to alleviate these problems:

1) "Divide and conquer". Now, the abstraction is implemented in a cluster in terms of a flexible representation, described totally by axioms, and any fixed representation satisfying these axioms may be used. Now, the proof consists in obtaining two mappings: from the fixed to the flexible representation and from the flexible representation to the abstraction. Each individually is probably easier to find than the one required for the "normal" methodology. In addition, if the fixed representation used is chosen from a library, the first mapping need not even be considered, since the correctness of the fixed representation with respect to the axioms of the flexible representation has already been proved (*)

(*)Hopefully a fixed representation of a flexible representation is not entered the library unless it has been proved correct.

2) If only the fixed representation is changed, then only the correctness of the new representation with respect to the axioms of the flexible representation need be proved. The proof involving the mapping from the flexible representation to the abstractions is still valid. Furthermore, if the new fixed representation is chosen from a library, nothing new has to be proved at all.

In our example, the spanning cluster, which implements the abstraction "set of sets of integers" is implemented in terms of two flexible representation rep1 and rep2. Each repi implements one level of sethood, i,e.,

set of set of integer
repl (rep2 (integer))

and as such, each has identical semantics for the defining operations as given in the previous section.

There are several concreate representations for this flexible representation.

1. linked list with updates in place
2. varying array as with PL/1 CHAR VARYING [IBM72]
3. flexible rows as in Algol 68 [VWN74]
4. fixed array of sufficiently large size as in PASCAL[JW74]
5. files as in PASCAL[JW74]

Each is space or time efficient for different applications. In our case, it turns out that rep2 should be (4) and repl should be (1) [AHU74].

Consider now what would happen if the cluster used the fixed representations chosen above, i,e.

array [1..n] of linked-list of integer

In this cluster each of add, select, etc would have two different guises depending on which level set was being added to, selected, etc., and their identities as add, select, etc., would be lost. Specifically,

op	Array of	linked-list
add(s,'+',e)	shift all elements down by 1 and assign e to first position	cons
add(s,'-',e)	assign e to one more than last position	update nil of last cell to point to newly allocated cell containing e
Select (s,i)	subscript with i	step down list while counting to i

The same operations at each set level will have radically different code sequences, and we would end up proving the same behavior twice under two different guises. In using the flexible representations we may define the set operations at each level in terms of the same add, select, etc. The correctness of the implementation of the set operations can be proved at a higher level and only once with respect to the axioms of repi(t). Then we need only ascertain that the concrete representations chosen implement the flexible representation correctly, but if these concrete representations are from a library, the proof have already been performed.

How the Proof Was Done

To do the proof of correctness of the cluster spanning, first, mapping had to be obtained from the flexible representations to the sets represented.

1) A repl consisting of a sequence of integers represents a set of integers:

for $r \in$ repl(integer)
$A''(r)=\{j \mid r<j>\neq\Lambda\}$

2) Each repl is indexed within the sequence VS, which is a rep2 (repl), by an integer setname. It is useful to map from this setname to the set.

for $i \in$ int,
$A(i)=\{j \mid VS<i \circ j>\neq\Lambda\}$

3) VS, a rep2 (repl), represents a set of sets, specifically the set each whose elements is the set represented by an element of VS.

For VS $\in$ rep2 (repl (integer))= rep
$A'(vs)=\{A(j) \mid \exists j \ni VS<j>\neq\Lambda\}$

It is fairly straightfoward to prove that the bodies of replace-by-union and add modify VS in such a way that they implement the appropriate set-of-sets operations, i,e.,

1) replace-by-union(a,b),where a and b are set names, makes a name the union of the sets named by a and b, and b name the empty set.

   It is assumed that these sets are disjoint so that repetitions do not have to be checked for. It is easy to see that VS<a> and VS<b> are modified to reflect these changes under the mapping.

2) add(a), where a is an integer makes VS the union of the set of sets represented by VS and {{a}}. It is easy to show that VS is modified correctly

to reflect this change under the mapping.

However, find, which gives the name of the set containing its parameters, and #, which gives the cardinality of the set of sets represented by VS, are more difficult to deal with. If they were implemented by brute force algorithms:

1) find(e): search each VS<i> in order until VS<i∘j>= e end return i as the name of the set containing e.

2) # : counting up from 1, return the first i-1 such that VS<i>=Λ.

They could be proved correct directly from the mapping. However, the brute force algorithms are not efficient enough. It makes more sense, as we have done in the cluster, to keep some auxiliary information in some bookkeeping variables to allow find and # to be computed directly from the values of these variables. Hence,

1) VS<0> consists of a table mapping an element e to the name of the set currently containing e, i,e,

   VS<∘e> = name

2) number-of-sets contains the current cardinality of the set of sets represented by VS.

It is necessary, of course, that each operation that changes the set membership of an element or the number of sets in the set of sets update VS<0> or number-of-sets appropriately.

The fact that find and # return the correct values cannot be deduced directly from the mappings any more.

The solution is to express the constant relations between the values of the bookkeeping variables and the

represented sets in an invariant of the representations. This invariant includes the facts that

1) VS<0∘e> is the name of the set, if any, that contains e;
   if e is in no set then VS<0∘e> =0
2) number-of-sets equals the cardinality of A'(VS)

The invariant will also express the relation between the represented sets and any other variables introduced for the convenience of the implementation.

The invariant of the representations must be shown to hold after VS is created and must be shown to be preserved through each operation (If an operation has no side effects, this is so trivially). The proof that find and #, which merely consult bookkeeping variables, work properly is trivial.

In the appendix, we lay the groundwork for the proof of correctness of the spanning tree cluster with respect to the usual behavior of sets of sets of integers. We define the mapping, give the invariant of the representations and then state the input and output assertions that the operation bodies must satisfy in order for the cluster to be correct. Finally, we give annotated copies of the operation bodies showing that they do in fact satisfy the required input and output assertions.

6. CONCLUSIONS

We have discussed how the methodology and features of PEP support portability and efficiency. These properties can be achieved in our system if each machine for which we compile a PEP program hosts a library of fixed representations in its own machine language. Based in such a library a programming system can automatically select the cost efficient fixed representation for a particular use of a set of pre-defined standard operations.

We have also illustrated how the methodology supported by PEP's features help in discovering proofs of correctness of a data type implementation, by enforcing a modular implementation.

We hope that the design principles we experimented with in PEP were able to illustrate how software engineering principles can be used as a matrix for programming language design.

APPENDIX

a. MAPPINGS

The representation mappings obtain from an integer set name, the integer elements of the named set, from the VS variable, the set of sets of integers represented and from a list of integer elements, the set containing those elements.

A: int $\rightarrow \mathcal{P}$ (int)

A (i) = {j | VS<i, j> $\neq \Lambda$}

A': rep $\rightarrow \mathcal{P}$ ($\mathcal{P}$(int))

A'(VS) = {A(VS<j>) | A(VS<j>) $\neq$ {}}

A'': repl (integer) $\rightarrow \mathcal{P}$(int)

A'' (k) = {i | $\exists$ j $\ni$ k<j> $\neq \Lambda$}

b. INVARIANT OF THE DATA REPRESENTATION

The invariant of the representation captures the assumed property that each element from 1 through n is in at most one set plus some properties relating the various bookkeeping variables, i.e. size, number-of-sets, name-generator, and VS<0> to the represented set of sets.

We define a number of simple assertions and give names to them. The invariant of the representation, INVREP, is then defined as the conjunct of these simple assertions. In all of the following i,s,t $\in$ int.

DISJOINT $\triangleq$ ($\forall$i) (1$\leq$i$\leq$n $\supset \exists$ at most one s $\ni$ 1<s$\leq$name-generator $\wedge$ i $\in A$(s))

In this assertion and in others, we consider set names only up to name-generator, because at any time name-generator contains the name of the last set of integers which was allocated.

(*) SIZESETSOK $\triangleq$ ($\forall$s) (1$\leq$s$\leq$name-generator $\supset$ card (A(s)) = size (s))

(*) card (set) is the number of elements in the set

SIZEVSOK $\triangleq$ card ($A'(VS)$) = number-of-sets

NAMEGENOK $\triangleq$ number-of-sets + card ({s| 1≤s≤name-generator} $\wedge A(s)=\{\}$)= name-generator

NAMESOK $\triangleq$ setname (i) = s iff
[s = 0 $\supset$ ($\exists$t) (1≤t≤name-generator $\wedge$ i $\in$ A(t))
$\wedge$ s $\neq$0 $\supset$ i $\in A(s)$]

INVREP $\triangleq$ DISJOINT $\wedge$ SIZESETSOK $\wedge$ SIZEVSOK $\wedge$ NAMEGENOK $\wedge$ NAMESOK

c. STATEMENT OF LEMMAS TO BE PROVED

We are now in a position to state the input and output assertions with respect to which the operation bodies must be partially correct in order for the cluster to be correct.

Create must initialize VS so that it represents the empty set of sets and it must set the auxiliary variables to make the INVREP true.

True: {Q create} $A'(VS) = \{\} \wedge$ INVREP

Add takes as its parameter an integer e and updates VS to represent the union of its current value and the set {e}.

Of course auxiliary variables must be reset to preserve the INVREP.

$A'(VS)= X \wedge e \in$ int $\wedge$ INVREP {Q add}
$A'(VS)= X \cup \{\{e\}\} \wedge$ INVREP

Replace-by-union takes two integer set name parameters a and b and fixes things up so that b names the union of the two sets named by a and b, a names the empty set, and the bookkeeping variables reflect the one less set of integers and the new set sizes for a and b.

It is assumed that before the operation, a and b both name non-empty sets.

$$A(a) = S1 \wedge A(b) = S2 \wedge S1 \neq \{\} \wedge S2 \neq \{\} \wedge \text{INVREP } \{Q \text{ replace-by-union}\}$$

$$A(a) = \{\} \wedge A(b) = S1 \cup S2 \wedge \text{INVREP}$$

Find finds the name of the set containing its parameter as an element. It is assumed that find is passed only elements that are in some set.

$i \in \underline{int} \wedge 1 \leq i \leq n \wedge (\exists s)(1 \leq s \leq \text{name-generator} \wedge i \in A(s)) \wedge \text{INVREP } \{Q \text{ find}\}\ i \in A(\underline{return}) \wedge \text{INVREP}$

\# simply returns the cardinality of the set of sets represented by VS.

$$A'(VS) = x \wedge \text{INVREP } \{Q\#\} A'(VS) = x \wedge \underline{return} = card(x) \wedge \text{INVREP}$$

ANNOTATED PROGRAMS

Instead of giving complete proofs of the partial correctness of the operation bodies with respect to their input and output assertions, which would be only a tedious illegible mess, we give annotated versions of the code for the operations. The annotations, consisting of assertions and other comments, are inserted at sufficient and strategic points that the corresctness of the code with respect to the assertions is "perfectly clear".

In the following:

belongs $(b\langle i\rangle, a) \triangleq$ setname $(VS\langle b \circ i\rangle) = a$

belongs $(b[i \ldots j], a) \triangleq (\forall m)(1 \leq m \leq j \supset \text{belongs}(b\langle i\rangle, a))$

For sets S1 and S2

min $(S1, S2) \triangleq$ *if* card (S1) > card (S2)
then S2 else S1 *i-end*

```
max (S1,S2)≜ if card (S1)>card(S2)
             then S1 else S2 i-end

create

    ¢ true ¢
    number-of-sets ←0;
    name-generator ←0;
    for n times do VS<s>$add('-',0)f-end
    ¢ A'(vs)={} ∧ INVREP ¢
    ¢ since none of VS<i>≠Λ, A'(VS)={}. Most of INVREP holds
      vacuously and/or by inspection ¢

endcreate

    find: op (i:integer)returns (integer);
    ¢ i∈int ∧ 1≤i≤n  ∧ (∃s) (1≤s≤name-generator ∧
    i ∈A(s)) ∧ INVREP ¢

return   (VS<0 ∘i>)

    ¢ i ∈A(return)∧ INVREP ¢
    ¢ since VS<0 ∘i> = setname(i) and

    INVREP ⊃NAMEOK and NAMEOK ∧(∃s)(1≤s≤ name-generator ∧
    i∈ A(s) ⊃ i ∈A(setname(i))¢

end find;

add: op (e:integer)

      ¢A'(VS)= x ∧ e ∈ int ∧ INVREP ¢
      name-generator + ←1;
      ¢A'(VS)= x ∧ e∈ int ∧ INVREP  name-generator number-of-set
                                    name-generator-1 number-of-sets-1 ¢

      size(name-generator)← 1;
      VS$add('-',Λ);
      VS<name-generator>$add(e);
      ¢ A'(VS)= x ∪ {{e}} ∧  SIZESETSOK ∧ DISJOINT ∧  NAMEGENOK ∧  ¢
              NAMESOK  name-generator
                       name-generator-1 ¢
```

```
      VS$replace(0 ∘ e,name-generator)
      ¢A'(VS)= x ∪ {{e}} ∧ INVREP ¢
end add;
# : op () returns(integer);
      ¢A'(VS)=x ∧ return= card(x) ∧ INVREP ¢
      ¢ follows directly from SIZEVSOK of INVREP ¢
end #;
replace-by-union: op (a, b: integer), id K, element:
integer:
      ¢A (a)=S1 ∧A(b)=S2 ∧ S1 ≠ {} ∧ S2 ≠ {} ∧ INVREP ∧
      if size(a)>size(b) then a↔b i-end;
      ¢A(a)= min (S1,S2)∧A(b)= max (S1,S2)∧ S1 ≠ {} ∧ S2 ≠ {} ∧
                                                  INVREP ¢
      element←VS<a ∘1>; k←1;
      ¢ loop invariant: A(a)=min (S1,S2)∧A(b)=max (S1,S2)
      ∧ S1 ≠ {} ∧ S2 ≠ {} ∧ DISJOINT ∧ SIZESETOK ∧ SIZEVSOK ∧
      NAMEGENOK ∧ NAMESOK  s≠ 0
                           s≠0∧s≠ a

      (element ≠ Λ ⊃ belongs (a[1..k), b)∧ belongs(a[k..size
      (a)],a))

      ∧(element=Λ ⊃ k=size(a)+1 ∧ belongs(a[1..size(a)],b)) ¢

      while element ≠ Λ do
            VS$replace(0 ∘element,b);
            k+←1;
            element← VS<a,k>
      w-end
      ¢ A(a)=min (S1,S2)∧ A(b)= max (S1,S2)∧ S1 ≠ {}∧
         S2 ≠ {} ∧ DISJOINT ∧ SIZESETSOK ∧ SIZEVSOK ∧
         NAMEGENOK ∧ NAMESOK  s≠ 0      ∧
                              s≠0∧s≠ a
```

```
    belongs (a[ 1..size(a)] ,b),¢
    VS<b>$add('-',VS<a>);
    VS$replace(a,Λ);
¢ A (a)={} ∧ A(b)=S1 ∪ S2 ∧ DISJOINT ∧ NAMESOK ∧

(SIZEVSOK ∧NAMEGENOK) number-of-sets
                      number-of-sets-1   ∧

                ∀s
SIZESETSOK                ∧ size(b)= card (A(b))- size(a) ¢
                ∀s≠b

size(b)←size(a) + size(b);

¢ since DISJOINT: size (b)= card (A(b))¢
number-of-sets-1 - ← 1
¢ A(a)={}∧ A(b)= S1 ∪ S2 ∧INVREP ¢

end replace-by-union;
```

REFERENCES

AHU74 AHO, A.V.; HOPCROFT, J.E.; ULLMAN, J.D. - The design and analysis of computer algorithms. New York, Addison - Wesley, 1974.

CAR75 CARVALHO, S. et alii - An overview of PEP language: a language for portability, efficiency & provability. (To appear in PUC's Monograph series)

EAR73 EARLEY, J. - Relational level data structures for programming languages. Acta Informatica, 2, 1973.

GOT74 GOTTLIEB, C.C. & TOMPA, F.W. - Choosing a storage schema. Acta Informatica, 3, 1974

HOA72 HOARE, C.A.R. -Proof of correctness of data representations. Acta Informatica, 1, 1972.

IBM72 PL/1 language reference manual. New York, IBM, 1972. Form nº GC28-8201-4.

JEW74 JENSEN, K. & WIRTH, N. - PASCAL - user manual and report. Berlin, Springer Verlag, 1974.

LIS74 LISKOV, B. it alii - CLU design notes. Cambridge , Mass., MIT Project MAC, 1974.

LIZ74 LISKOV, B. & ZILLES, S. - Programming with abstract data types. In: SIGPLAN symposium on very high level languages, march, 1974.

LIZ75 LISKOV, B. & ZILLES, S. - Specification Techniques for data abstractions. IEEE Transactions on Software Engineering, 1 (1), 1975.

LOW74 LOW, J.R. - Automatic coding choice of data structures. Stanford, Stanford University, Computer Science Department, 1974. STAN-CS-74-452.

MLM74 MELKANOFF, M.; LAUTERBACH, G; MOSZKOWSKI, B. - Implementation of MADCAP-6 on UCLA'S 390-91 IMLAC System. Los Angeles, University of California, School of Engineering, 1974. UCLA-ENG-7438.

MOR72 MORRIS, J.B. & WELLS, M.B. - The specification of program flow in MADCAP-6. SIGPLAN Notices, 7 (11), 1972.

MUG73 MULLISH, H. & GOLDSTEIN, M. - A SETLB primer. New York, New York University, Courant Institute of Mathematical Sciences, 1973.

NAU63 NAUR, P. et alii - Revised report on the algorithmic language Algol 60. Numerische Mathematik, 4, 1963.

SCH72 SCHWARTZ, J.T. - Abstract algorithms and a set theoretic language for their expression. New York, New York University, Courant Institute of Mathematical Sciences, 1971.

SCH72 SCHWARTZ, J. T. - Principles of specification on language design with some observations concerning the utility of specification languages. In: COURANT Computer Science Symposium, 4. Englewood cliffs, Prentice-Hall, 1972.

STA73 STANDISH, T - Data structures: an axiomatic approach. Cambridge, Mass., Bolt, Beranek & Newman , 1975.

TOM75 TOMPA,F.W. - Evaluating the efficiency of storage structures. Waterloo, University of Waterloo, Dept. of Computer Science, 1975. CS-75-16

VWN74 VAN WIJNGAARDEN, A. et alii - Revised report on the algorithmic language Algol 68. Edmonton, University of Alberta, 1974.

ERRATA

Pages	Lines	Where it reads	Please, read
1.	12	abststractions we propose...	abstractions. We propose...
3.	18	subtracting components, etc...	subtracting components, selecting components, etc...

5. The attached page (5) should replace the original one.

Pages	Lines	Where it reads	Please, read
6.	4	VS's down...	VS is down...
6.	8	are different then in...	are different, then in...
6.	22	line 13...	line 19...

Pages

7., 8. The attached pages (7,8) should replace the original ones.

Pages	Lines	Where it reads	Please, read
10.	7	for "select"),...	for select),...
10.	13	(lines 14-2)...	(lines 14-21)...
10.	18	VS<a.K>...	VS<a.K>...
10.	30	up to date...	up-to-date...

Pages

11., 12. The attached pages (11,12) should replace the original ones.

Pages	Lines	Where it reads	Please, read
13.	23	cost function. (paragraph)	cost function. The table... (without paragraph)
15.	last	Specifically,	Specifically:
16.	next to last	proof have already...	proofs have already...
17.	3	mapping had to...	mappings had to...
17	7	for R∈ repl (integer)	for R∈ repl (integer)
17	17	For VS... (integer))...	For VS... (integer))
Pages	Lines	Where it reads	Please, read
19.	23	annotated...	asserted...
19.	31	based in...	based on...
19.	32	cost efficient...	cost-efficient...
20.	2	help in discovering...	helps in discovering...
20.	7	as a matrix...	as a metric...

Pages

21.-26. The APPENDIX attached (21-26) should replace the original one.

```
1     minimum-cost-spanning-tree: proc (reads n: integer) ;
2     type arcs: set of edges; type vertices: set of vertex;
3     type edge: tuple of vertex;
4     type vertex: subrange (1..n);
5     type spanning: set of (set of vertex);
6     id E,T: arcs;
7     id VS←{}: spanning; ¢initializing declaration ¢
8     id W1, W2, V: vertices;
9     id v,w : vertex;
10    read  E and V;
11    T←{} ;
12    for v in V do VS← VS ∪ {v}
13    while card(VS)>1 do
14       pick one  <v,w> in E such that
         (∀<s,t> in E) (cost (<s,t>)≥ cost(<v,w>));
15       E←E - {<v,w>} ;
16       W1←unique s in VS such that v in s ;
17       W2←unique s in VS such that w in s ;
18       if W1 ≠ W2 then  begin
19          replace W1 and W2 by W1 ∪ W2 ;
20          T←T ∪ {<v,w>}
21         end
22       i-end
23    w-end
24    p-end  minimum-cost-spanning-tree
```

Figure 1

Three sets are used: E which contains the edges of the graph, V which contains the vertices of the graph and T which is used to collect the edges of the minimum cost spanning tree. The algorithm transforms a spanning forest into a single tree according to the cost function. The set contains the trees in the spanning forest.

Initially, the forest contains one tree for each vertex consisting of only that vertex. Each cycle through the loop expands one tree to include the vertices of another and reduces the size of the forest VS by one.

```
0   minimum-cost-spanning-tree: proc (reads n: integer) ;
1   type arcs: set of  edges; type vertices: set of vertex;
2   type  edges: tuple of vertex;
3   type vertex: subrange (1..n);
4   type spanning: set of (set of vertex) with cluster (n);
5   id n : integer;
6   id E, T: arcs;
7   id VS: spanning; ¢ initializes VS to {} ¢
8   id W1, W2, V: vertices;
9   id v,w: vertex;
10  read E,V, and n;
11  T←{};
12  for v in  V do VS$add(v);
13  while  VS$#  > 1 do
14        pick one  <v,w> in E such that
          (∀<s,t> in E)(cost(<s,t>)≥ cost(<s,w>));
15        E←E - {<v,w>};
16        W1← VS$find(v);
17        W2← VS$find(w);
18        if W1≠W2 then   begin
19            VS$replace-by-union(W1,W2);
20            T←T ∪ {<v,w>}
21            end
22        i-end
23  w-end
24  p-end minimum-cost-spanning-tree;
```

Figure 2a.

```
1       spanning: cluster (n: integer) on rep2(rep1 (integer))
        is replace-by-union, find, add, # ;
2       id size: integer  array [1:n] ;
3       id VS: rep;
4       id number-of-sets: integer;
5       id name-generator : integer;
6       create
7          number-of-sets← 0;
8          name-generator← 0;
9,10       for n times do VS<0>$add('-',0) f-end
11      endcreate ;
12      replace-by-union: op (a, b : integer );
13         id k, element: integer
14         if size (a) > size (b) then a↔b i-end ¢ see footnote ¢
15         element ← VS<a∘1>;
16         k ← 1;
17         while element ≠ Λ do
18            VS$replace (0∘element, b);
19            k + ←1; ¢ see footnote  ¢
20            element ← VS<a∘1> ;
21         w-end
22         VS<b>$add ('-' VS <a>);
23         VS$replace(a,Λ);
24         size (b) ← size (a) + size (b) ;
25         number-of-sets-←1;
26      end replace-by-union
27      find: op (i:integer) returns (integer);
28      return (VS<0∘1>);
29      end find;
```

Figure 2b

"k+←1" means "k←k+1" . Likewise for -← etc;

"a←→b" means "exchange values of a and b".

4. PORTABILITY

Our approach to portability is based on the idea of standardizing the instruction set that operates on the flexible representation. This notion was suggested by the work of Standish [STA73] where an attempt is made to axiomatize the basic properties of all data structures.

Assuming a storage structure that behaves like Standish's data spaces we have converted the selection and assignment operations into a set of convenient basic operations that form the instruction repertoire of our cluster level (i.e. of our flexible representation). These basic operations have a very simple and universal semantics which does depend on the actual implementation of the flexible representation.

Let $r \in rep(t)$. Then a value of r is a sequence of t's selected by consecutive integers starting from 1. The length of such a sequence is one less than the first index j such that the selection of the j^{th} element yields Λ. The operations assumed are

add: rep(t)x{'+','-'}x t → rep(t)
add: (S,p,e)$\underline{\underline{\Delta}}$ add e to the beginning or end
(depending on whether p is'+' or '-') of S
sub: rep(t) x t → rep(t)
sub: (S,e)$\underline{\underline{\Delta}}$ remove e from S if e is in S
select: rep(t)x int → t
select: (S,i)= S<i> $\underline{\underline{\Delta}}$ the i^{th} element of S if it exists and Λ otherwise
replace: rep(t)x int x t → rep(t)
replace: (S,i,e)$\underline{\underline{\Delta}}$ change the i^{th} element of S to e if S<i>$\neq\Lambda$
insert: rep(t) x int x {'+','-'} x t →rep(t)
insert: (S,i,p,e)$\underline{\underline{\Delta}}$ insert e into S before or after (depending on whether p is '+' or'-') the i^{th} elements of S.

In the above, whenever an element is inserted or removed the indices are shifted to preserve the fact that consecutive integers from 1 through the length select non-Λ elements. Formally, we have the following axioms:

Let r = rep (t)

1) $v \in r = [(\forall i)(1 \le i \le \text{length}(v) \supset (\text{select}(v,i) \in t \wedge \text{select}(v,i) \neq \Lambda))]$

2) $v = \text{add}(s,'+',e)$ iff $[\text{length}(v) = \text{length}(s)+1 \wedge \text{select}(v,1) = e \wedge ((\forall i)(1 \le i \le \text{length}(s) \supset \text{select}(v,i+1) = \text{select}(s,i)))]$

3) $v = \text{add}(s,'-',e)$ iff $[\text{length}(v) = \text{length}(s)+1 \wedge \text{select}(\text{length}(v)) = e \wedge ((\forall i)(1 \le i \le \text{length}(s) \supset \text{select}(v,i) = \text{select}(s,i)))]$

4) $v = \text{sub}(s,e)$ iff
$(\ j \quad e = \text{select}(s,j) \supset$
$[\ \text{length}(v) = \text{length}(s-1) \wedge$
$[(\forall i)(1 \le i \le j-1 \supset \text{select}(v,i) = \text{select}(s,i))$
$\wedge((\forall i) \quad (j+1 \le i \le \text{length}(s) \supset$
$\text{select}(v,i-1) = \text{select}(s,i)))] \wedge$
$((\exists j) \ni e = \text{select}(s,j)) \supset v = s$

5) $v = s$ iff $(\forall j)(\text{select}(v,j) = \text{select}(s,j))$

6) $v = \text{replace}(s,i,e)$ iff
$((\text{select}(s,i) \neq \Lambda) \wedge (j \neq i \supset \text{select}(v,j) = \text{select}(s,j)) \wedge \text{select}(v,i) = e)$

7) $v = \text{insert}(s, i, '+', e)$ iff $[1 \le i \le \text{length}(s) \supset (\text{length}(v) = \text{length}(s)+1 \wedge ((\forall j)(1 \le j \le i) \supset \text{select}(v,j) = \text{select}(s,j)) \wedge ((\forall j)(i+1 \le j \le \text{length}(s)) \supset \text{select}(v,j+1) = \text{select}(s,j)) \wedge \text{select}(v,i+1) = e)]$

8) $v = \text{insert}(s,i,'-', e) \supset$ iff $[1 \le i \le \text{length}(s) \supset (\text{length}(v) = \text{length}(s)+1 \wedge ((\forall j)(1 \le j \le i-1) \supset \text{select}(v,j) = \text{select}(s,j)) \wedge ((\forall j)(i \le j \le \text{length}(s)) \supset \text{select}(v,j+1) = \text{select}(s,j)) \wedge \text{select}(v,i) = e)]$

Notation: v<i>= select(v,i)

APPENDIX

a) MAPPINGS

The representation mappings obtain from an integer set name, the integer elements of the named set, from the VS variable,the set of sets os integer represented and from a list of integer elements, the set containing those elements.

$$A: \underline{int} \rightarrow \mathcal{P}(\underline{int})$$

$$A(i)=\{j \mid VS<i \circ j>\neq \Lambda\}$$

$$A: \underline{rep} \rightarrow \mathcal{P}(\mathcal{P}(\underline{int}))$$

$$A'(VS)=\{A(VS<j> \mid A(VS<j>)\neq\{\}\}$$

$$A'': repl(integer) \rightarrow \mathcal{P}(\underline{int})$$

$$A(k)=\{i \mid \exists j \ni k<j>\neq\Lambda\}$$

To define the invarint of the representation it will be convenient to be able to obtain the integer name of the set that contains a given integer element. This function is defined only if there is a unique such set.

setname: int → int

$$setname(c)=\iota i \ni \exists\ j \ni VS<i \circ j>=c$$

b) INVARIANT OF THE DATA REPRESENTATION

The invariant of the representation captures the assumed property that each element from 1 through n is in at most one set plus some properties relating the various bookkeeping variables, i.e. size, number-of-sets, name-generator, and VS<0> tothe represented set of sets.

We define a number of simple assertion and give names to them . The invariant of the representation, INVREP, is then defined as the conjunct of these assertions. In all of the following i,s,t $\in$ int.

$$\text{DISJOINT} \triangleq (\forall i)(1 \le i \le n \supset \exists \text{ at most one } s \rightarrow 1 < s \le \text{name-generator} \wedge i \in A(s))$$

In this assertion and in others, we consider set names only up to name-generator , because at any time name-generator contains the name of the last set of integers which was allocated.

(*) $\text{SIZEZETSOK} \triangleq (\forall s)\ (1 \le s \le \text{name-generator} \supset \text{card}\ (A(s)) = \text{size}\ (s))$

$\text{SIZEVSOK} \triangleq \text{card}(A'(VS)) = \text{number-of-sets}$

$\text{NAMEGENOK} \triangleq \text{number-of-sets} + \text{card}\ (\{s \mid 1 \le s \le \text{name-generator} \wedge A(s) = \{\}\}) = \text{name-generator}$

$\text{NAMESOK} \triangleq \text{setname}\ (i) = s$ iff
$[s = 0 \supset (\neg \exists t)\ (1 \le t \le \text{name-generator} \wedge i \in A(t))$
$\wedge\ s \neq 0 \supset i \in A(s)]$

$\text{INVREP} \triangleq \text{DISJOINT} \wedge \text{SIZESETSOK} \wedge \text{SIZEVSOK} \wedge \text{NAMEGENOK} \wedge \text{NAMESOK}$

c) STATEMENT OF LEMMAS TO BE PROVED

We are now in a position to state the input and output assertion with respect to which the operation bodies must be partially correct in order for the cluster to be correct.

Create must initialize VS so that it represents the empty set of sets and it must set the auxiliary variables to make the INVREP true.

.... ..

true: $\{\Omega\ \text{create}\} A'(VS) = \{\} \wedge \text{INVREP}$

(*) card (set) is the number of elements in the set

Add takes as its parameter an integer e and updates VS to represent the union of its current value and the set {e} . Of course auxiliary variables must be reset to preserve the INVREP.

$$A'(VS)=X \wedge e \in \underline{int} \wedge INVREP \ \{Q\ add\}$$
$$A'(VS)=X \cup \{\{e\}\} \wedge INVREP$$

Replace-by-union takes two integer set name parameters a and b and fixes things up so that b names the union of the two sets named by a and b, a names the empty set, and the bookkeeping variables reflect the one less set of integers and the new set sizes for a and b. It is assumed that before the operations, a and b both name non-empty sets.

$$A(a)=S1 \wedge A(b)=S2 \wedge S1\neq\{\} \wedge S2\neq\{\}$$
$$\wedge\ INVREP\ \{Q\ \underline{replace\text{-}by\text{-}union}\}$$
$$A(a)=\{\} \wedge A(b)=S1 \cup S2 \wedge INVREP$$

Find finds the name of the set containing its parameter as an element. It is assumed that find is passed only elements that are in some set.

$$i\in\underline{int} \wedge 1\leq i\leq n \wedge (\exists s)(1\leq s\leq \text{name-generator} \wedge i\in A(s))$$
$$\wedge\ INVREP\ \{Q\ find\}\ i\in A(\underline{return}) \wedge INVREP$$

simply returns the cardinality of the set of the sets represented by VS.

$$A'(VS)=x \wedge INVREP\{Q\ \#\}\ A'(VS)=x \wedge \underline{return} = card\ (x)$$
$$\wedge\ INVREP$$

ANNOTATED PROGRAMS

Instead of giving complete proofs of the partial correctness of the operation bodies with respect to their input and output assertions, which would be only a tedious illegible mess, we give annotated versions of the code for the operations. The annotations, consisting of assertions and other comments, are inserted at sufficient and strategic points that the correctness of the code with respect to the assertions is "perfectly clear".

In the following:

belongs (b <i>,a) $\triangleq$ setname (VS<b∘i>)=a

belongs (b[i...j],a) $\triangleq$ (∀m) (i≤m<j ⊃ belongs(b<i>,a))

belongs (b[i...j],a) $\triangleq$ (∀m) (1≤m≤j ⊃ belongs (b<i>,a))

For sets S1 and S2

min (S1,S2) $\triangleq$ if card (S1)> card (S2)
then S2 else S1 i-end

max (S1,S2) $\triangleq$ if card (S1)> card (S2)
then S1 else S2 i-end

create

¢ true ¢
number-of-sets ←0;
name-generator ←0;
for n times do VS<s>$add('-',0)f-end
¢ A'(vs)={} ∧ INVREP ¢
¢ since none of VS<i>≠Λ, A'(VS)={}. Most of INVREP holds vacuously and/or by inspection ¢

endcreate

find: op (i:integer) returns (integer);
¢ i∈int ∧ 1≤i≤n ∧ (∃ s) (1≤s≤name-generator ∧ i∈A(s)) ∧ INVREP ¢
return (VS<0∘i>)
¢ i∈A(return) ∧ INVREP ¢
¢ since VS<0∘i> = setname(i) and
INVREP ⊃ NAMEOK and NAMEOK ∧ (s) (1≤s≤ name-generator ∧
i∈A(s) ⊃ i∈A(setname(i))¢

end find;
add: op (e:integer)
¢ A'(VS)=x ∧ e∈int ∧ INVREP ¢
name-generator + ←1;
number-of-sets + ←1;

¢ A'(VS)=x ∧ e∈int ∧ INVREP$^{\text{name-generator number-of-set}}_{\text{name-generator-1 number-of-sets-1}}$ ¢

```
    size(name-generator) ← 1;
    VS$add('-',Λ);
    VS<name-generator>$add(e);
    ¢ A'(VS)=x∪{{e}} ∧ SIZESETSOK ∧ DISJOINT ∧ NAMEGENOK ∧ ¢
                        name-generator
    ¢       NAMESOK                          ¢
                  name-generator-1
    VS$replace(0∘e,name-generator)
    ¢ A'(VS)=x∪{{e}} ∧ INVREP  ¢
end add;

# : op () returns(integer);
    ¢ A'(VS)=x ∧ INVREP  ¢
    return(number-of-sets)
    ¢ A'(VS)=x ∧ return= card(x) ∧ INVREP ¢
    ¢ follows directly from SIZEVSOK of INVREP ¢
end #;

replace-by-union: op (a,b: integer), id k, element: integer;
    ¢ A(a)=S1 ∧ A(b)=S2 ∧  S1 ≠  {} ∧ S2 ≠ {} ∧ INVREP ¢
    if size(a)<size(b) then a↔b i-end ;
    ¢ A(a)= min(S1,S2) ∧ A(b)= max(S1,S2) ∧ S1 ≠ {} ∧ S2 ≠ {} ∧ INVREP ¢
    element←VS<a∘1> ; k←1;
    ¢ loop invariant: A(a)=min (S1,S2) ∧ A(b)=max (S1,S2)
    ∧ S1 ≠  {}  ∧ S2 ≠ {} ∧  DISJOINT ∧ SIZESETOK ∧ SIZEVSOK ∧ NAMEGENOK
              S ≠0
    ∧ NAMESOK        ∧
              S ≠0∧S≠a
    (element≠Λ ⊃belongs (a[1..k], b) ∧ belongs (a[k..size(a)],a))

    ∧(element=Λ ⊃k=size(a)+1 ∧ belongs(a[1..size(a)],b)) ¢

    while element ≠ Λ do
          VS$replace(0∘element,b);
          k+←1;
          element←VS<a∘k>

    w-end;
    ¢ A(a)=min(S1,S2) ∧ A(b)=max(S1,S2) ∧ S1 ≠ {} ∧
       S2 ≠ {} ∧ DISJOINT ∧ SIZESETSOK ∧ SIZEVSOK ∧ NAMEGENOK
                 s≠0
       ∧ NAMESOK        ∧
                 s≠0∧≠a
```

```
        belongs(a[1..size(a)],b) ¢
        VS<b>$add('-',VS<a>);
        VS$replace(a,Λ);

    ¢ A(a)={} ∧ A(b)=S1 ∪ S2 ∧ DISJOINT ∧ NAMESOK ∧

    (SIZEVSOK ∧ NAMEGEBOK)  number-of-sets
                            number-of-sets-1^

              ∀s
    SIZESETSOK      ∧ size(b)= card (A(b))- size(a) ¢
              ∀s≠b
    size(b)←size(a) + size(b);
    ¢ since DISJOINT: size(b)= card (A(b)) ¢
    number-of-sets ← -1.
    ¢ A(a)= {} ∧ A(b)= S1 ∪ S2 ∧ INVREP  ¢
end replace-by-union;
```