PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 14/76

DESIGN AND IMPLEMENTATION OF A DATA ABSTRACTION DEFINITION FACILITY

by

Daniel Schwabe
and
Carlos J. Lucena

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Marquês de São Vicente 225 — ZC 19
Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 14/76

DESIGN AND IMPLEMENTATION OF DATA ABSTRACTION DEFINITION FACILITY*

by

Daniel Schwabe
and
Carlos J. Lucena

Series Editor: Roberto Lins de Carvalho October 1976

* This work has been supported in part by FINEP under contract Nº CT/370.

ABSTRACT:

This paper describes the implementation model of a data definition facility for abstract data types, implemented as an extension to PL/I. The facility is based on a modified version of the cluster mechanism for the implementation of types. The proposed version tries to address the issues of efficiency and portability in connection with the goal of systematic programming.

KEYWORDS:

Data types, data abstractions, cluster, data representation, implementation model, PL/I.

RESUMO:

Este trabalho descreve um modelo de implementação de um mecanismo para a definição de dados, orientado para tipos abstratos de dados, que foi implementado em PL/I. O mecanismo é baseado numa versão modificada da técnica de "clusters" usada para a implementação de tipos. A versão proposta procura tratar os tópicos de eficiência e portabilidade, associados com o objetivo de programação sistemática.

PALAVRAS CHAVE:

Tipos de dados, abstração de dados, "cluster", representação de dados, modelo de implementação, PL/I.

CONTENTS

1. INTRODUCTION

Most of the current research in support of Software Engineering is oriented toward the creation of better tools for programming. A major emphasis is being placed on designing new programming languages to make the task of programming more "natural" and systematic [WIR 71, DDH 72] .

Abstraction plays a major role in programming and is the basis of the most important tools available to the programmer for control of the complexity of programs. One may say that "structured programming" is essentially based on the notion of appropriate selection and use of abstractions.

For the above reasons it is desrible that a programming language be designed to support the use of abstractions and thereby create a linguistic level in which it is more "natural" for the programmer to structure his algorithms.

One attempt in this direction has evolved from the cluster approach of Liskov and Zilles [LIZ 74, LIS 74] based on the class concept in SIMULA 67 [DMN 68, DAH 72]. In our work, we describe a variation of the cluster approach that supports the use of abstractions and also addresses issues of portability and efficiency.

In the following section we review briefly the cluster approach. Section 3 describes our proposed extension to clusters. The remaining section describes a system implemented in PL/I to support one proposed extension by providing a data definition facility for abstract types.

2. THE CLUSTER APPROACH

Abstract data types (or simply types) are groups of objects together with the operations applicable to them: A cluster is defined as an independent external module that initializes and manipulates the representation of an object of type T. A cluster contains the definition of the (concrete) representation of the object of the type, as well as the bodies of the procedures used to initialize and modify this representation so as to reflect the abstract operations defined for the type.

The general form of a cluster is

```
T: cluster <cluster formal parameter list option> is p1,p2,...,pm ;
   rep = <template>
   create  <formal parameter list option>
      Q0
   end

P : operation <operation formal parameter list option>
    <return type option>;
      Q1
    end
    :
Pn: operation ...
      Qn
    end
end T
```

where T is the name of the cluster (and abstract type) being defined; $p_1,\ldots,p_n$ are the names of the valid operations defined for type T; the <template> mentioned in the rep statement is the actual representation used for this type (which may be a primitive type of the language, another cluster type, or even a type built out of the type rep, allowing for recursive data types). Q_0 is the body of the create operation, which is executed whenever a program using a type T requests the creation of an object of this type. $Q_1,\ldots,Q_n$ are the bodies of

the operations defined for type T . The parameters of the cluster are other types, so the cluster is actually a type constructor; the parameters of the create operation are initial values for the representation, For more information on the meaning of the various parameter list options and return type options see [LIS 74].

3. AN EXTENSION TO THE CLUSTER CONCEPT

The extension proposed requires the decomposition of a cluster into two parts. We separate the concrete representation of the type from the cluster itself. In the resulting first part (called general representation level cluster), the user describes the effects of the operations defined for the abstract type through a generic representation that refers to a common base representation (defined in terms of standard operations).

The second part (called the concrete representation level cluster) makes use of the "lowest level" features of the host language to implement as efficiently as possible the common base representation (i.e. the standard operations) in terms of basic data structures.

Each (new) type of cluster is still an independent external module; nevertheless, concrete representation level clusters may be used only inside general representation level clusters, which, in turn, may use other general representation level clusters.

A complete program can then be described in three levels (which are similar to Earley's [EAR 73])

At the first level (called the specification level),which is analogous to Earley's relational level,an algorithm is constructed using basic (primitive) types in the language, as well as abstract types. At the second level, which is analogous Earley's access path level, the abstract types used in the previous level are described in terms of a common base representation. At the lowest level,which is analogous to Earley's implementation level, the common base representation is described in terms of the host language.

This approach has several advantages over the traditional one:

(i) A data type is often described in terms of an access path to a representation (for example, stacks and queues of objects). In the traditional approach, the user has to describe the same access path in several different ways, one for every concrete representation used, even though the "abstract" behavior of the access mechanism is the same. In the present model, the general representation level cluster describes only this "abstract" behavior.

(ii) Since the general representation level clusters do not make any assumption about the concrete representation, they constitute better modules for the decomposition of larger programs; the common base language that refers to the concrete representation level serves as an adequate programming mechanism for the achievement of modular programming, as discussed in [DEN 75b].

(iii) Abstract machine modelling is a well known technique for achieving portability [POW 75]. The concrete level of the representation can be regarded as defining an abstract machine. Since a compiler is able to generate code for both the abstract and primitive data types used in a program (expressing it in terms of the primitive commands that handle the common base representation) a high degree of portability can be obtained.

(iv) By gathering run time statistics about a program and attributing cost functions to the standard operations implemented through concrete representation level clusters (in a way similar to [GOT 74, TOM 75, LOW 74]),it is possible to select the best concrete representation cluster for a given application. In fact, it is not difficult to envisage a system in which this selection is done automatically from a library of concrete representation clusters (as in [LOW 74]). Thus, run-time efficiency can be predicted and controlled in a machine-independent fashion.

(v) The situation mentioned in (i) above is also reflected in proofs of correctness of the referred clusters - one has to prove the same "abstract" behavior several times, once for each concrete repre - sentation used. It is expected that the mappings used in proving

the correctness of the original concept of cluster [BEL 76] will be decomposed into simpler mappings under the proposed extension. This concept was first introduced in the language PEP [LSB 75].

4. PL/I IMPLEMENTATION OF DATA DEFINITION FACILITY

A common approach when experimenting with a new language feature is to embed it in an existing language. The developers can then benefit from the criticisms of the established community of users of that language. With that purpose in mind, an extension to PL/I was developed [SCH 76].

At the specification level, three additional statements were made available in PL/I:

(i) For the system to recognize an identifier as the name of an abstract type the user must write

```
DCL  T  ABSTRACT_TYPE;
```

(ii) To declare a variable V of type T, declared above, the user must write (the use of parameters will be explained later)

```
DCL V T (parameters);
```

(iii) Let us suppose that op is the name of a valid operation for type T. To apply this operation to a variable V the user must write

```
... T@op(V,parms)...
```

where parms stands for the actual parameters required by operation op.

As an example, we present a program to solve the "sequence" problem. This problem was proposed by Wirth in [WIR 71] It can be stated as follows: "Construct a sequence using of length N the set {1,2,3} such that there are no two adjacent equal subsequences".

The solution is based on the fact that a valid sequence of length L must contain two overlapping valid sequences of length L-1.

Starting from an empty candidate sequence we extend it by appending a 1 to it. If the sequence formed in this way is not valid, the last digit of the sequence is incremented by one; if it is a 3, it is necessary to truncate the sequence at the last digit different from 3, increment it by one and then check it again. These operations are repeated until the length of the candidate sequence reaches N. The following program illustrates a solution for the proposed problem.

```
GENERATE: PROC OPTIONS(MAIN);
 DCL   CAN_SOL ABSTRACT_TYPE;
 DCL   X       CAN_SOL(PIC'9');  /*INITIALIZES X TO THE EMPTY
                                   SEQUENCES */

 DCL   N   BIN FIXED;
 GET   LIST(N);
 DO WHILE(CAN_SOL@LENGTH(X)¬=N);
    CANDID@EXTEND(X);
    DO   WHILE(¬CAN_SOL@OK(X));
         CAN_SOL@CHANGE(X);
    END;
 END;
 PUT SKIP EDIT('SEQUENCE FOUND:',CAN_SOL@SYMBOL(X)) (A,A);
END GENERATE;
```

In this system, the common base language operates on a representation; this representation is considered to be an indexed sequence of values of the form $\Gamma = \Gamma_1\Gamma_2 \ldots \Gamma_n$. The following standard operations are allowed:

ADD: adds an element to either end of the sequence
SUB: subtracts an element from either end of the sequence
SELECT: selects an element anywhere in the sequence
INSERT: inserts a new element " " " "
REMOVE: removes an element " " " "
REPLACE: replaces an existing element anywhere in the sequence
LINK: links two sub-sequences
DETACH: detaches two sub-sequences

COPY: generates a copy of the sequence
LENGTH: provides the cardinality of the sequence

The index of an element is always relative to the beginning of the sequence it is in.Thus the index of an element well be changed after performance of an operation which has a side effect on its sequence argument.

A concrete representation level cluster has the following general form:

```
C : REP USES <templates>;
    [declaration of the variables of the representation]
    CREATE (size);
       body of CREATE operation
    ENDCREATE;
    ADD: PROC (POS,ELEM);
        body of standard operation ADD
    END ADD;
    SUB : PROC(POS,ELEM)
        :
    END SUB;
    SELECT : PROC(POS) RETURNS(POINTER);
        :
    END SELECT;
        :
    LENGTH : PROC;
        :
    END LENGTH;
END C;
```

Where:

(i) <template> stands for a PL/I based variable (or aggregate);

(ii) among the global variables, there must be a pointer variable that

points to the last allocation of the variable used as <template>; only scalar variables can be global;

(iii) each standard procedure always has the same formal parameters; their meaning can be deduced from the examples;

(iv) the parameter size of the create operation is the initial length of the representation.

Since the concrete representation may be used to store values of any type (including other representations), it was decided that they should actually store pointers to the values. This approach has the disadvantage of leaving the manipulation of the values themselves to the general representation cluster, but it is a simple way a achieving the desired generality. Furthermore, through this approach, if the value stored is actually an array, structure or any other data aggregate, assigning it becomes very simple.

The CREATE operation is executed whenever a concrete representation is required. The following concrete representation level cluster implements a one way linked list.

```
LIST: REP USES
     1 NODE BASED(PT),
       2 VALUE POINTER,
       2 NEXT  POINTER;
     DCL (HEAD,LAST) POINTER, SIZE BIN FIXED;
     CREATE(S);
       DCL(I,S) BIN FIXED, AUX POINTER;
       IF S > 0 THEN DO
                     ALLOCATE NODE SET(PT);
                     AUX,HEAD = PT;
                     END;
       DO I = 1 TO S - 1;
         ALLOCATE NODE SET(PT);
         NODE.VALUE=NULL;
         AUX → NODE.NEXT = PT;
         AUX = PT ;
       END
```

```
  IF S > 0 THEN LAST = PT;
           ELSE LAST = HEAD;
  SIZE = S;
  PT → NODE.NEXT = NULL;  /*LAST NODE OF THE LIST*/
ENDCREATE;
SELECT: PROC(I) RETURNS(POINTER);
    DCL (I,J) BIN FIXED, (K,M) POINTER;
    K = NULL;
    IF I > SIZE THEN RETURN (K);
    K = HEAD → NODE.NEXT;  /*GETS THE FIRST ELEMENT OF THE LIST*/
    DO J = 1 TO I WHILE(K = NULL); /*STEP THROUGH THE LIST TO THE*/
      M=K;
      K=K→NODE.NEXT;
    END;
    IF J = I+1 THEN DO;
                    PUT SKIP LIST('ERROR');
                    STOP;
                    END;
    RETURN(M→NODE.VALUE);
END SELECT;
LENGTH: PROC RETURNS(BIN FIXED);
  RETURN(SIZE);
END LENGTH;
ADD: PROC(POS,ELEM);
    DCL POS CHAR(1), ELEM POINTER;
    ALLOCATE NODE SET(PT);
    PT→NODE.VALUE = ELEM;
    PT→NODE.NEXT  = NULL;
    IF POS = '+'
      THEN  DO          /*PUT THE NEW ELEMENT AT THE END OF THE
                          LIST*/
            LAST NODE.NEXT = PT;
            LAST = PT;
           END;
      ELSE IF POS = '-' /*PUT THE NEW ELEMENT AT THE
                        /* BEGINNING OF THE LIST*/
             THEN DO;
                  PT→NODE.NEXT = HEAD
                  HEAD = PT;
```

```
                    END;
                ELSE DO;
                    PUT SKIP LIST('PARAMETER ERROR');
                    STOP;
                    END;
          SIZE=SIZE+1;
        END ADD;
        SUB : PROC...
            .
            .
            .
        END SUB;
        .
        .
        .
    END LIST;
```

The set of PL/I procedures and variables declarations generated from the concrete representaion cluster makes use of much of the code in the cluster the name of each standard procedure is prefixed with the name of the cluster to form the name of the corresponding PL/I procedure, e.g., SELECT becomes LISTSELECT. The header of this cluster is transformed into a declaration of the <template>. Since the global variables are also part of the representation, they are gathered in a PL/I based structure. In the previous example, the following declaration would be generated

```
DCL  1  LISTAUX,BASED($2), 2((HEAD,LAST)POINTER,SIZE BIN FIXED);
```

A variable of abstract type used at the specification level will in fact contain a pointer to this auxiliary structure (remember there is always a variable in this structure that points to the data structure built with the template, e.g. HEAD or LAST).

All standard procedures have,as an implicit parameter, a pointer to an instance of this auxiliary structure (and therefore to the data structure used). This parameter is inserted in the parameter list of all standard procedures, with the name '$R'. All ocurrences of global variables are then prefixed by '$R→', e.g. M = HEAD; becomes M = $R→HEAD;.

The general representation cluster has the following form:

```
T: CLUSTER (formal parameters) ON  <representation level indicator> IS
  [(] op1(list of types of parameters) <return option> [)],...
  [(] opn(list of types of parameters) <return option> [)];
      declaration of variables global to the cluster
   CREATE
      body of CREATE operation
   ENDCREATE
op1: PROC(parameter list) <returns option>;
     body of operation op1
END op1;
   .
   .
   .
opn: PROC...
   .
   .
   .
END opn;
END T;
```

Where:

(i) The formal parameters to the cluster stand for any primitive type in PL/I or any constant. They cannot be declared inside the cluster; they are actually macro parameters and are substituted through macro expansion.

(ii) The symbol<representation level indicator>stands for a sequence of symbols of the form REP1 [(REP2 ...)]. There are certain kinds of abstract data types that have an access path in more than one level (e.g., set of sets of integers, hashtables with collisions placed in a list). This component of the header is used to indicate how many levels the representation has; a different concrete representation may be associated with each level.

(iii) The list of types of parameters for each operation in the header is the same type of list used for the ENTRY attribute allowed in standard PL/I. The only additional type of parameter is the REP parameter which stands for a parameter of the abstract type. The first parameter in all operations must always be of type REP. If the declaration of an operation in the header is enclosed in parentheses, this operation is regarded as internal to the cluster, i.e., it may be used only

within other operations in that cluster.

(iv) The return option is the same as the standard one in PL/I, except for the fact that, when an operation returns an object of the abstract type, the type of the value returned is declared as REP.

(v) In addition to scalar variables of any primitive type in PL/I, variables of any of the following types may be global in the cluster:

(a) template variables - these variables are used to describe the contents of the representation (remember the representations contain pointers to the values). A template variable is any PL/I based variable; to declare it, one uses the syntax of the DECLARE statement in PL/I, with the keyword TEMPLATE substituted for DECLARE ou DCL e.g., TEMPLATE A(20) BIN FIXED BASED (PT_A);

(b) variables of abstract types and abstract type identifiers.

(c) representation variables - these variables are used when it is necessary to reference a "generic" concrete representation. To declare a variable as being of representation type it is necessary to use the attribute REP_j, where j is an integer that stands for the level (as explained in (ii) above). For example, it is possible to write DCL R REP1(1); - the parameter is the initital length of the representation.

(vi) The bodies of the operations may contain any type of statement mentioned previously, plus references to the standard operations for concrete representations.

The CREATE block contains code to initialize global variables as well as the (generic) representation used. Therefore, this block must contain at least the declaration of a representation variable to stand for the generic representation used. In the case where the type being defined implies an access path of more than one level, this variable will be at the outermost level. This variable is an implicit parameter to all operations in the cluster, and should not appear in the parameter list of any of those operations.

It should be noted that values of variables of abstract type may also be stored in the representation (since they are actually pointers). Nevertheless, a special syntax is required for that. It is presented later in the paper.

As an example of a general representation cluster, one show how the abstract type candidate used previously can be implemented:

```
CAN_SOL: CLUSTER(TYPE) ON REP1 IS
           EXTEND(REP), CHANGE(REP), OK(REP) RETURNS(BIT(1);
           LENGTH(REP) RETURNS(BIN FIXED),
           SYMBOL(REP) RETURNS(CHAR(200) VAR);
       TEMPLATE $N TYPE BASED(PTN);
       TEMPLATE $M TYPE BASED(PTM);
   CREATE;
     DCL X REP1(0);        /*THE SEQUENCE IS INITIALLY EMPTY*/
   ENDCREATE;
   EXTEND: PROC;
       ALLOCATE $N;
       $N = 1;
       REP ADD(X,'+',PTN); /*NOTICE THAT A POINTER TO THE VALUE
                             IS STORED*/
       RETURN;
   END EXTEND;
   CHANGE: PROC;
     DCL N1 TYPE;
   L=REP@LENGTH(X);
   PTN=REP@SELECT(X,L);     /*GET THE LAST ELEMENT OF THE SEQUENCE*/
   REP@SUB(X,'+');          /*AND REMOVE IT*/
   DO WHILE($N=3);          /*IF IT WAS A 3, THE SEQUENCE MUST BE*/
     FREE PTN→$N;           /*TRUNCATED*/
     L=REP@LENGTH(X);
     PTN=REP@SELECT(X,L);
     REP@SUB(X,'+');
   END;
   N1=$N+1;                 /*INCREASE THE LAST ELEMENT BY 1*/
   ALLOCATE $N;
   $N=N1;
   REP@ADD(X,'+',PTN);
```

```
        RETURN;
    END CHANGE;
        :
END CAN_SOL;          /*NOT ALL OPERATIONS ARE SHOWN*/
```

In the same way as before, the names of the operations in the cluster are prefixed with the name of the cluster when translated into PL/I. The header of the cluster is transformed into a series of declarations with the entry attribute for the operations.

Since the standard operations are the same, no matter what concrete representation is used, it is impossible to determine (when translating a general representation cluster) which actual operations is to be called when a reference to a standard operation is made. In fact , this will depend on the representation used for its first argument, and may vary from call to call within the same program. Therefore, it is necessary to introduce an argument for each operation in the cluster to identify which concrete representation cluster is being used for the first argument. This identification is then passed to an interface program (at run time) that consults it and calls the appropriate concrete representation cluster operation. Since a variable may be of a type that has an access path in more than one level, it is possible to have more than one concrete representation associated with each variable. For that reason the identification must be a vector of integer numbers.

Each concrete representation, when entered into the system library, receives an unique identification number. This number is used to initialize an identification vector associated with each variable of abstract type.

Evidently, the representation variable declared in the create block is also included as a parameter for every operation in the cluster. As an example, the header of the operation extend in the previous example would be expanded into

```
EXTEND: PROC($ID,$R);
```

where $ID is the identification vector, and $R is the representation variable that stands for the actual parameter, which is of abstract type.

When a variable is declared as being of representation type, a call is made to the appropriate create operation (through the interface program) in the concrete representation cluster. For that reason, this type of declaration is in fact an executable statement. In the example , the statement DCL X REP1(0); is translated into

```
DCL X POINTER;
X = CREATE($ID,0);
```

where $ID is the identification vector that is passed as a parameter to the create block where the original declaration appears.

When a variable of abstract type is used as a parameter (either formal or actual) to an operation other than those defining its type, it is necessary to indicate that by prefixing it with a '.', so that additional information is included (identification vector, etc...).

In order to be rigorous, the global variables of the general representation cluster mus be part of the representation of a variable that uses that cluster. For this reason, these variables are gathered in a based structure much in the same fashion as in concrete representations cluster, and a pointer to it is also included in the value of a variable of a type modeled by that cluster.

5. CONCLUSION

In summary, the basic techniques used in the system can be described as follows. For each variable declared as being of abstract type at the specification level, there are three pieces of information associated to it:

(i) An identification vector containing the unique identification number(s) used for that variable. This is specified in a kind of "JCL" command issued before the translation actually starts;

(ii) A pointer to a structure containing the values of all the variables that are global to the cluster used;

(iii) A pointer to a structure containing the values of all the variables that are global to the concrete representation used. If there is more than one concrete representation being used (access path of more than one level), this refers to the outermost level.

At the point of declaration at the specification level, a call is made to the create operation in the corresponding cluster; this initializes the pointer described in (i) above. This create operation in turn, contains of the concrete representation cluster, and this mechanism initializes the pointer described in (ii). Note that the declaration of a variable to be of some abstract type is also an executable statement.

The program implementing this extension is a preprocessor written is SPITBOL and occupies around 35K (generated code) on a IBM 370/165-OS-MVT. Running in a 150K partition (which actually leaves only 25K of available dynamic core for execution), the preprocessor took slightly less time than the PL/I F compiler to process the sample programs. As concrete representation clusters are put into a library, one would expect the preprocessing time to drop. The complete sequence program was (pre) processed in 3.9 seconds.

Experience thus far has shown increase in memory usage, although only simple problems were run. The interface program (which is an almost fixed part in all programs using this system) uses around 1900 bytes of code, which a significant amount in the sample programs. That should not happen in larger programs.

A similar reasoning applies to the concrete representation clusters - in some of the examples, they were too sophisticated for the needs of the algorithms, because they used too much code. The one-way linked list implementation of the concrete representation used 1309 bytes of code; the whole program generated used 7786 bytes, with a (static) data area of 3308 bytes.

An interesting situation occurred while debbuging the example presented in this paper. The first errors we had involved mainly pointers, and were due to programming errors in the concrete representation cluster. As soon as these were corrected, the program output a sequence, but composed only of 1's. This turned out to be an error in the operation in the CAN_SOL cluster that checked for the validity of a sequence. Finally, when this was fixed up, we got a correct sequence but of length N+1 instead of N, and this was caused by an incorrect test in the main loop of the specification program. This shows clearly how the errors moved upwards through the several levels. This observation gives an indication that this approach might ease the task of programming, specially when clusters are taken from a library of supposedly correct clusters.

REFERENCES

DEN 75a - Dennis, J.B. - The Design and Use of Software Systems - in Software Engineering, An Advanced Course, Lecture notes in Computer Science, nº 30, Springer Verlag, 1975.

DEN 75b - Dennis, J.B. - Modularity, Software Engineering, An Advanced Course, Lecture notes in Computer Science,nº30,Springer Verlag,1975.

WIR 71 - Wirth, N. - Program Development by Stepwise Refinement - CACM 14(5), May 1974.

DDH 72 - Dahl, O.J., Djkstra, E.W.; Hoare, C.A.R. - Structured programming - Academic Press, 1972.

DAH 72 - Dahl, O.J., Hoare, C.A.R. - Hierarchical Program Structures - in DDH72.

DMN 68 - Dahl, O.J.; Myhrhang, B., Nygaard, N. - The SIMULA 67 Common Base Language, Oslo, Norwegian Computer Centre, 1968.

LIS 74 - Liskow, B., Ziller, S. - Programming with Abstract Data Types - SIGPLAN Symposium on Very High Level Languages, March, 1974.

LIS 74 - Liskov, B., A Note on CLU - CLU Design Notes, MIT Project MAC, November, 1974.

EAR 73 - Earley, J. - Relational Level Data Structures for Programming Languages - Acta Informatica 2, 1973.

POW 75 - Poole, P.C.; Waite, W.M. - Portability and Adaptability - in Software Engineering - An Advanced Course, Lecture notes in Computer Science, nº 30, Springer Verlag, 1975.

GOT 74 - Gotlieb, C.C.; Tompa, F.W. - Choosing a Storage Schema - Acta Informatica 3, 1974.

TOM 75 - Tompa, F.W. - Evaluating the Efficiency of Storage Structures - University of Waterloo, Dept of Computer Science CS-75-16, 1975.

LOW 74 - Low, J.R. - Automatic Coding: Choice of Data Structures - Stanford University, Computer Science Dept., STAN-CS-74-452, 1974.

BEL 76 - Berry, D.M.; Erlich, Z.; Lucena, C.J.P. - Structured Data Representations - Proposed Modifications to the concept of Cluster - Proceedings of the SIGPLAN Conference on Data Abstractions, Definition and Structure, Salt Lake City, March 1976.

LSB 75 - Lucena, C.J.P., Schwabe, D.; Berry, D.M. - Issues in Data Type Construction Facilities - PUC Technical Report nº 4/75.

SCH 76 - Schwabe, D. - Aspectos de Engenharia de Software no Projeto de Linguagens de Programação (Software Engineering Issues in Programming Language Design) - Pontifícia Universidade Católica, 1976 - M.Sc. Thesis.