SPECIFICATION AND UNIFORM REFERENCE TO DATA STRUCTURES IN PL/I

Daniel Schwabe
and
Carlos J. Lucena
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro - Brasil

Through the use of a modified concept of cluster [1, 2] we propose the association of the notions of abstract data types and uniform reference to data structures to PL/I. The proposed programming mechanisms enhance PL/I by the addition of two new linguistic dimensions: a specification level and a common base language to handle the implementation of data structures. This report informally describes the syntax and semantics of the added constructs and gives an example of their use.

INTRODUCTION

An effective way of testing a new programming mechanism is to embed it in a well-known programming language. Instead of forming a significant number of users in yet another completely new programming language, it seems better to evaluate a new modeling capability (programming construct) through its presentation in the context of a very familiar notation (a frequently used language). Our research group has repeatedly used this approach with encouraging results [3, 4, 5].

The features we propose in the sequel were first introduced in an altogether innovative language design [6] and then transferred into PL/I for the purpose of testing.

The cluster approach is due to Liskov and Zilles [1] 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 needs to be concerned only with the abstract behavior of the type as presented by the operations.

The model we propose requires the decomposition of a program into three levels (as in [7]):

i. At the first level, the user describes (specifies) an algorithm in a very high level notation implied by the possibility of defining abstract data types (either off-the-shelf or custom tailored abstraction).

ii. At the second level the user describes the access mechanisms involved in the so-called general level of the representation which is chosen to implement the abstract type (cluster type 1). The general level of the representation is defined in terms of a set of standard operations which provide a uniform way of handling most implementation level (concrete) data structures (still undefined at this point).

iii. At the third level the standard operations on the general representation are modelled by a different cluster (type 2) that makes use of the "lowest level" features of the host language. This is the implementation or concrete level of the representation.

We have argued elsewhere [2] that from the points of view of provability, efficiency and portability, the above approach has many advantages.

2. THE NOTATION AT THE DIFFERENT LINGUISTIC LEVELS

An abstract type, in our extended version of PL/I, is declared in the following way.

```
DCL name (parameter list) ABSTRACT TYPE;
```

A variable can then be defined by writing

```
DCL var name;
```

Whenever op is a valid operation (defined for type name), it can be applied to a variable of this type by writing

```
name@op (var, parameter list);
```

A cluster that defines the general level of the representation (access path level) has the following form (cluster type 1):

```
name: CLUSTER ON REP1 (parameter list) IS op$_1$,...,op$_n$;
      [declaration of global (to the cluster) variables]
      CREATE
         DCL r REP;
         [create body]
      ENDCREATE
      op$_1$: PROC (parameter list)  RETURNS (type);
         [declaration of local variables]
         [op$_1$ body]
      END op$_1$;
      .
      .
      op$_n$: PROC...
      .
      .
END name;
```

At this level all the primitive PL/I data types can be used with the exception of pointer and based variables. This is meant to delay the use of implementation details to the concrete representation level[9]. The CREATE block (procedure) is activated by the declaration DCL var name; used at the specification level.

In the definition above we wrote DCL r REP; to mean that r is of whatever concrete representation is used (recall that an arbitrary concrete representation is operated upon by a set of standard operations). The set of standard operations on a concrete representation are used in the definition of the semantics of the op$_i$ ($1 \le i \le n$). When the abstract type being modelled implies more than one level access path (e.g. sets of sets of integers) the header of the cluster must indicate that in the following form

```
name: CLUSTER ON REP$_i$ (REP$_{i-1}$(...(parameter list)...)) IS op$_1$,...,op$_n$;
```

In that case, r will stand for the outermost REP.

The declaration of global and local variables may contain the definition of abstract types, thus making general representation clusters accessible from within other general representation clusters.

The operations in the general representation clusters may be defined over two (or more) variables of the same type: a typical example is the assignment operation . In this case, we require that the concrete representations used for each argument be the same. In this respect we follow Low [8].

A cluster that defines the concrete representation level (type 2) has the following form:

```
repr: REP (parameter list) USES <template> ;
          [declaration of global (to the cluster) variables]
        CREATE
          [create body]
        ENDCREATE
        ADD: PROC (parameter list);
          [declaration of local variables]
          [body of standard operation ADD]
        ENDADD;
        :
        SUB: PROC...
        :
        SELECT: PROC...
        :
      END repr;
```

At the above representation level the programmer can make use of full-PL/I. The symbol <template> stands for the PL/I data types used to implement the concrete representation. We have defined the following set of standard operations:

ADD (adds an element to the defined in terms of the <template>)

SUB: subtracts an element
SELECT: selects an element
INSERT: inserts a new element

REPLACE: replaces an old element
LINK: links two sub-structures
DETACH: detaches two sub-structures
COPY: generates a copy of the structure
SUCC: finds the successor of a given element
PRED: finds the predecessor of a given element

These operations can be easily axiomatized as in [2].

The CREATE block initializes the concrete representation. In the general representation level, the declaration of any variable as being of type REP (or REP_i) causes the activation of the corresponding CREATE block of the concrete representation cluster.

To allow for more flexibility, clusters of the second and third levels may have parameters. These parameters contain the basic (primitive to PL/I) types of which the ABSTRACT_TYPE is formed. This is reasonable since the general representation cluster describes an access path that is independent from the types of the elements in accesses. Evidently, these types must be passed to the concrete representation cluster, since it is there that these types will ultimately appear.

3. HINTS ON THE TRANSLATION TO STANDARD PL/I

In this section we give a brief description of how an equivalent set of programs in standard PL/I can be obtained from our proposed extension.

Before the translation actually starts, it is necessary to associate a concrete representation cluster to every variable in the specification level that is declared as being of ABSTRACT TYPE. This association is accomplished by the statement

```
ASSIGN REP repr TO var
```

The idea is to have each variable of ABSTRACT_TYPE actually declared as a pointer variable; this variable will point to an instance of the concrete representation that is constructed through the operations in the general representation cluster (type 1). Since all concrete representation clusters have the same standard operations, the actual procedures are distinguished by prefixing the cluster name to the operation. A call to these operations uses an interface program that in turn calls the actual operation in the concrete representation cluster being used. Thus,

```
DCL var name;
```

translates into

```
DCL var POINTER;
var = nameCREATE;
```

and

```
:
name@op(var,parameters)
:
```

generates

```
          CALL nameop(var,b,parms);
```

The parameter b is an integer used to identify the concrete representation cluster. This integer number is unique to each of these clusters.

Each operation in the general representation cluster, such as

```
          op: PROC (parms)RETURNS (REP);
```

is translated to

```
          name op: PROC(r,s,parms) RETURNS (POINTER);
              DCL r POINTER,s BIN FIXED;
```

Inside a cluster, a mention of any of the standard operations such as r@ADD(parms) is translated to CALL ADD (r,s,parms). In this case, s is the integer that will distinguish the appropriate operation in the concrete representation cluster. For each variable that uses this concrete representation the assigned unique integer is passed as a parameter in calls to the operations of this cluster.

The declaration

```
          estr: REP (parms) uses <template>;
```

is translated into

```
          DCL <template> BASED (ptestr);
```

and each of the standard operations will be prefixed by the name of the cluster, e.g., estrADD (r,s,parms). In addition, appropriate information is included in the interface program.

The body of the CREATE block (which is translated into a function) must contain an ALLOCATE statement for the template. Also, since all global variables in the concrete representation cluster are part of the structure, they are gathered in a PL/I BASED structure.

All these points will be illustrated in the following example.

4. EXAMPLE

The following sample program uses an abstract type stack (the favorite example of most of the authors in this area).

The main program is presented in the sequel.

```
EX:  PROC OPTIONS (MAIN);
      DCL stack(type ABSTRACT_TYPE;
      DCL p stack(BIN FIXED);
          .
          .
          .
      IF input='(' THEN stack@push(p,k)
                ELSE IF input=')' THEN DO;
                                     PUT SKIP LIST(k,stack@top(p));
                                     stack@pop(p);
                                     END
```

This would be part of a program that prints pairs of positions of parentheses in a string. The general representation cluster implementing a stack would be

```
stack: CLUSTER ON REP1 (type) IS push,pop,top;
   CREATE
       DCL r REP;
   ENDCREATE
   push : PROC (elem);
          DCL elem type;
          REP@ADD(r,'-',elem);
          RETURN
   END    push;
   pop  : PROC;
            REP@SUB(r,'-');
           RETURN
   END  pop;
   top  : PROC RETURNS(type);
           RETURN(REP@SELECT(r,0));
   END top;
END  stack;
```

In this cluster, ADD, SUB and SELECT refer to operations in a concrete representation level. It is clear that push, pop and top use the (fixed) semantics of the former operations. Finally, we show part of a concrete representation cluster that implement a linked list.

```
list   : REP(type) USES
      1 node,
        2  value    type,
        2  next     POINTER;
  DCL (head,last) POINTER,
          size       BIN FIXED;
  CREATE
      ALLOCATE node SET(head);
      node.value=0;
      node.next =NULL;
      last       =head;
      size       =0;
  ENDCREATE
```

```
    SELECT: PROC(i) RETURNS(type);
        DCL (i,j) BIN FIXED;
        DCL (k,m) POINTER;
        IF i > size THEN RETURN(UNDEF); /*UNDEF is undefined value*/
        k = head→node.next;
        DO j = 0 TO i WHILE(k¬=NULL); /*search for ith element*/
           m = k;
           k = k→node.next;
        END;
        RETURN(m→node.value);
    END SELECT;
    ADD : PROC(pos,elem);
        ECL pos CHAR(1),elem type;
        DCL pt pointer;
        ALLOCATE node SET(pt);
        pt→node.value = elem;
        pt→node.next  = NULL;
        size = size + 1 ;
        IF pos = '+'    /* File in the last position*/
         THEN DO;
               last→node.next=pt;
               last=pt;
              END;
         ELSE IF POS = '-'
         THEN DO;  /* File in the first position*/
         head→node.next=head;
               head=pt;
              END;
         ELSE CALL ERROR;
         RETURN;
    END ADD;
      .
      .
      .
END list;
```

The variable UNDEF stands for a representation of an undefined value. Assuming that we want to use a list to implement the stack, our specification level program should be preceded by

```
                ASSIGN REP list TO p
```

Suposing that list has an identification number 1 (which means that all calls refering to it will contain an 1B argument) the following translations would be generated in the specification level program:

```
EX: PROC OPTIONS(MAIN);
    DCL p POINTER;
    p=stackCREATE(1B);
      .
      .
      .
    IF input='(' THEN CALL stackpush(p,1B,k);
                 ELSE IF input=')' THEN DO;
                                      PUT SKIP LIST(k,stacktop(p,1B));
                                      CALL stackpop(p,1B);
                                      END
         .
         .
         .
END EX;
```

In the general representation cluster for stack, push would appear as

```
         .
         .
         .
         stackpush: PROC(r,s,elem);
            DCL r POINTER,s BIN FIXED;
            CALL ADD(r,s,'-',elem);
            RETURN;
        END stackpush;
         .
         .
         .
```

Finally, we show part of the translated concrete representation cluster.

```
       list: PROC;
          DCL
             1 node BASED($1)
               2  value  BIN FIXED,
               2  next   POINTER;
          DCL
             1 aux   BASED($2)
               2 head      POINTER,
               2 last      POINTER,
               2 size      BIN FIXED;
          listCREATE: PROC(b) RETURNS(POINTER);
               DCL pt POINTER,b  BIN FIXED;
               ALLOCATE aux SET(pt);
               ALLOCATE node SET(head);
                   .
                   .
                   .
               RETURN (pt);
         END listCREATE;
         list SELECT: PROC(r,s,i) RETURNS(BIN FIXED);
              DCL r POINTER, s BIN FIXED;
              DCL (i,j) BIN FIXED;
              DCL (k,m)  POINTER;
              DCL  temp  POINTER;
              IF i >r→size  THEN RETURN (UNDEF);
              temp=r→head;
              k = temp→node.next;
              DO j=0 to i WHILE(r¬=NULL);
                 m=k;
                 k=k→node.next;
              END;
              RETURN(m→node.value);
          END  listSELECT;
            .
            .
            .
END list;
```

In this procedure, the pointer variable temp is used because PL/I does not allow expressions like r→head→node.next.

REFERENCES

[1] Liskov, B.; Zilles, S. - Programming with Abstract Data Types - in SIGPLAN Symposium on very high level languages. March, 1974.

[2] Lucena, C.J.;Schwabe, D.; Berry, D. - Issues in Data Type Construction Facilities - Technical Report nº 4/75 - Pontifícia Universidade Católica, Rio - August, 1975.

[3] Furtado, A.L.; Pfeffer, A. - Pattern Matching for Structured Programming in PL/I - Seventh Asilomar Conference on Circuits, Systems and Computers. 1973.

[4] Furtado, A.L.; Santos, C.S. - G/PL/I -Extending PL/I for Graph Processing Fourth Symposium on Computer and Information Science, 1972.

[5] Bauer, J.C.P.; Furtado, A.L. - Extending the Control Structures of PL/I - PUC Technical Report.

[6] Carvalho, S.; Lucena, C.J.; Schwabe, D.; Rosa, P. - An Overview of the PEP Language - a Language for Portability, Efficiency and Provability To appear.

[7] Earley, J. - Relational Level Data Structures for Programming Languages - Acta Informatica - 2, 1973.

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

[9] Berry, D.M. - Correctness of Data Representations: Pointers - Internal Memorandum 143, UCLA Computer Science Dept., 1975.