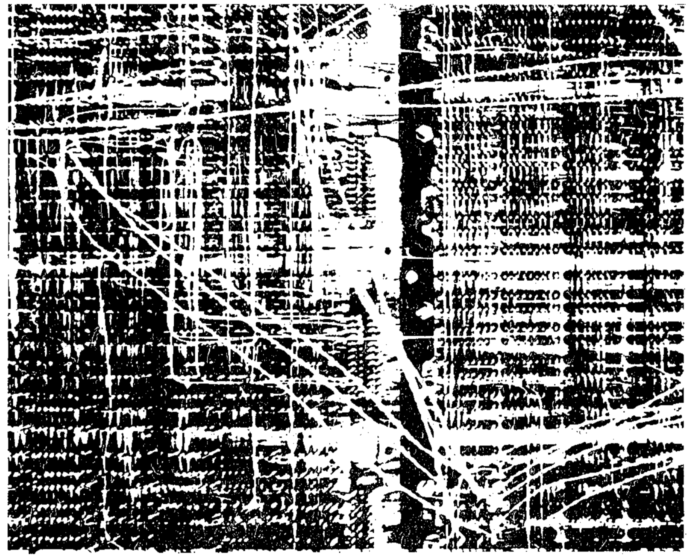

ACTAS DE LA **PRIMERA CONFERENCIA INTERNACIONAL EN CIENCIA DE LA COMPUTACION**

Auspician:
Banco de Chile •
Secico •
A.F.P. Santa María •
Leniz y Silva, •
Ingenieros Consultores
Cientec •
Cecinas Winter •
Revista Informativa •

PONTIFICIA UNIVERSIDAD
CATOLICA DE CHILE

UNIVERSIDAD DE CHILE

PRIMERA CONFERENCIA INTERNACIONAL
EN CIENCIA DE LA COMPUTACION

FECHA : 24 - 27 AGOSTO

LUGAR : CASA CENTRAL
PONTIFICIA UNIVERSIDAD
CATOLICA DE CHILE

Un Enfoque Formal para la Especificación y Diseño de Aplicaciones de Bases de Datos

Uma Aproximação Formal à Especificação e Desenvolvimento de Aplicações de Bancos de Dados

A Formal Approach to the Specification and Design of Database Applications

- J. M. V. de Castilho
- A. L. Furtado
- P. A. S. Veloso

Pontificia Universidade Católica
do Rio de Janeiro,
Brasil

Abstract

With the algebraic approach to abstract data types, data base applications can be formally specified, by describing the interactions among operations meaningful to the application area specialists in their own language.

The original specification covers the behavioral aspects of the data base application, and also, in a provisional way, other aspects such as accessibility, usage interface and representation. At later stages, the last three aspects are decoupled and refined, giving origin to a modular architecture. The modules provide set-structured access paths, interfaces for different classes of users, and representation by a version of the entity-relationship data model.

All modules are expressed in a procedural style of algebraic presentation, which is easy to translate into some symbol-manipulation language (SNOBOL, Icon, LISP, etc.). This leads to early testing and experimental usage, in addition to verifications of correctness.

Acknowledgment

We are grateful to M.A. Casanova for helpful suggestions.

1. *Introduction*

The main aspect stressed by the algebraic approach to the specification of abstract data types is the *behaviour* of the data type objects, as determined by the operations defined on them. This is of interest to data base practitioners since it opens the way to the formal specification of data base applications *using the same terminology of the applications*. In fact, as we show in this paper, fully formal specifications can be achieved without resorting to data models, thereby avoiding the bias that their early adoption may introduce.

Our treatment of the problem of formally specifying database applications is based on *canonical term algebras* [7], following the methodology proposed in [14]. We shall use the procedural style of algebraic specification that was introduced in [6]. A specification following this style can be easily translated into some symbol manipulation language, thereby making the specification executable [6,9] for experimental usage and testing.

Such approach may be used to treat other aspects of data base applications as well, of which we shall investigate *accessibility*, *usage interface* and *representation*. At later stages, these aspects can be decoupled from the original specification and refined, but the resulting modular architecture is required to preserve the behavior initially specified. Multilevel architectures are advocated in the data base area [1,5] and elsewhere [3].

For a comprehensive bibliography on the subject, the reader is referred to [2]. Further results are reported in [12,13,17].

2. *First stage : specification of the application*

As an example of a (simplified) data base application, we shall use throughout the paper the data base of an employment agency, where persons apply for positions, companies subscribe by offering positions, and persons are hired by or fired from companies. A person applies only once, thus becoming a candidate to some position; after being hired, the person is no longer a candidate but regains this status if fired. The same company can subscribe several times, the (positive) number of positions being added up. Only persons that are currently candidates can be hired and only by companies that have at least one vacant position. One consequence of these *integrity constraints* is that a person can work for at most one company.

Apply, *subscribe*, *hire* and *fire*, together with *initag* (which creates an initially empty agency data base) are our *update operations*. As *query operations* we shall use *iscandidate* and *worksfor*, which are predicates, and *haspositions*, which returns the number of unfilled positions in a company.

Any agency data base (agdb) object will be created through - and can therefore be denoted by - expressions involving applications of the update operations. It is possible to identify sets of expressions that denote the same agdb object, but one may choose representatives for each one of those sets, defining a convenient *canonical form* containing only some of the update operations, designated as *constructors* [8]. The constructors used to generate our canonical representatives will be operations *initag*, *apply*, *subscribe* and *hire*, arranged in the sequence :

$$hire^{i} (\ldots subscribe^{j} (\ldots apply^{k} (\ldots initag() \ldots) \ldots) \ldots)$$

where superscripts i, j and k denote that there are i hire's, j subscribe's and k apply's , i,j,k being non-negative. Occurrences of the same operation are ordered lexicographically with respect to their first argument (person for hire and apply, company for subscribe), in increasing sequence, from left to right; the order of execution is from the inside out: the first is initag and the last is the most external operation. For any person p, there can be at most one apply having p as argument; for any company c, there can be at most one subscribe having c as one of the arguments; if a person p and a company c appear as arguments of a hire, then p must appear in an apply and c in a subscribe; the number of appearances of a company c as argument of hire operations must not exceed the (positive) number m of positions offered in the subscribe corresponding to c.

A symbolic representation of a canonical representative is a *canonical term*. If agdb objects are represented by canonical terms we can specify the effect of applying one operation as follows (note how update and query operations affect each other):

a - Update operations map the set of canonical terms into itself. For example, a subscribe operation for a company that has already subscribed simply adds its number-of-positions argument to the number in the (single) subscribe for that company appearing in the canonical term; a fire operation cancels the corresponding hire in the canonical term. The application of an update operation may depend on conditions that can be checked through query operations. We adopted the decision that, whenever the conditions fail an update operation has no effect, i.e., it will yield as result the same canonical term supplied as argument.

b - Query operations yield a logical value, in the case of predicates, or some value obtained from components of the agdb object. They are executed by inspecting the canonical term supplied as argument.

Following the style in [6], figure 1 shows the procedural specification of the agency data base as a <u>data type module</u>. A striking difference between standard algebraic presentations and their procedural counterparts lies in the occurrence of "=>" instead of "="; the rewriting rules [10] embodied in the operations are now applied in a single direction. In order to improve readability the canonical terms are written using square brackets instead of parentheses and "|" instead of comma. The language features are self explanatory except, perhaps, for "?", which stands for any valid value of an argument, and "?<variable>" which, in addition, assigns the value found to a variable, as in PLANNER (see [18]).

If the operations in the expression below

```
fire(E3,C2,hire(E2,C2,hire(E1,C2,subscribe(C2,1,hire(E1,C1,
hire(E4,C1,apply(E1,hire(E3,C2,apply(E2,apply(E4,subscribe(C2,3,
apply(E3,subscribe(C1,2,initag()))))))))))))
```

are executed, the resulting canonical term is :

```
HIRE[E1 | C1 | HIRE[E2 | C2 | HIRE[E4 | C1 | SUBSCRIBE[C1 | 2 |
SUBSCRIBE[C2 | 4 | APPLY[E1 | APPLY[E2 | APPLY[E3 |
APPLY[E4 | INITAG ]]]]]]]]]
```

The aspects of accessibility, usage interface and representation, mentioned in the introduction, are covered in a rudimentary form in this original specification. As a preliminary phase in their application, the query and update operations are assumed to be able to access the relevant components of agdb objects. Usage interface is covered in that the operations supplied can be used as elements in a language for the manipulation of such objects. Finally, canonical terms are a form of representation for agdb objects.

However, the size and complexity of most data base applications require further development of the features of the specification that deal with the aspects above. In order to select (and sometimes order) the components to be accessed, we may need to create and maintain other structures of appropriate types, superimposed on the data base application, which share the components involved. These auxiliary structures are said to provide _access paths_. Since data base applications are handled by different classes of users with different needs and degrees of authorization, they must be given interfaces tailored to their distinct characteristics.

Most obviously, the representation of agdb objects by canonical terms must be replaced, perhaps through a series of levels, until some representation is obtained that can be implemented efficiently. This requires considerable effort that one is not willing to spend except for important or extraordinary applications. Hence, we should look for some _data model_, which we view as the most general (least restricted) member of a family of data base applications. Assuming that the data model has been effectively implemented, all we have to do is to build upon it the representation of our data base application.

3. _Later stages : decoupling and refinement_

We now develop a modular architecture, centered on the agdb data type module. The addition of modules for adequate accessibility, usage interface and representation should not disturb the original set of valid agdb objects.

3.1. _Access paths_

We shall use set-structured access paths. Since sets of elements are a well-known (parametric) data type, the respective data type module is not included (see [6]).

In our modular architecture the connection between two data type modules for the definition of access paths is done through a _transference module_, the operations of which are essentially a _composition_ of (in the example) query operations from the agdb data type and constructors from the set data type. Figure 2 shows one such operation - sempcomp - which gives the set of employees working for a company.

3.2. _Usage interfaces_

Usage interfaces are provided as _interface modules_, the operations of which are certain data base application operations, which can be restricted by incorporating further applicability conditions and extended by triggering other data base application operations [16].

Figure 3 shows one operation - C-hire - of the interface of a particular company C. The operation allows C to hire a person who has not applied to the agency, provided that at least 10 vacant positions will remain; as a triggered action, an apply is made on the person's behalf. In the case of less than 10 vacant positions, the simple hire operation is invoked.

3.3 Representation

As a data model we chose a version of the entity-relationship model [4,15], supporting only binary relationships and allowing atributes for entities but not for relationships.

The data model corresponds to the data type module (erm), shown in figure 4. The operations allow to create and delete entities within entity-sets, modify values of attributes ('*' stands for the undefined value) and link or unlink entities via a relationship. Corresponding query operations (all are predicates) are provided.

The connection between the data type modules (of the data base application and of the data model) defining the representation, is done through a representation module, partly shown in figure 5 (see also [8], pg. 75). The operations in representation modules are specified as the substitution of programs involving data model operations for each data base application operation.

The data model can be seen to be fully compatible with the data base application. Persons (candidates and employees) and companies are entities, number of positions offered is an attribute of companies and WORKS is a relationship between persons and companies. The basic integrity constraint of the data model - links can only be maintained if both entities linked exist (in at least one entity-set) - is complemented, but not contradicted, by the special constraints governing the WORKS relationship.

The proposed architecture (figure 6) can be further extended by incorporating other access paths (based, for example, on lists and mappings [11]), which can, in turn, be represented at lower levels. Of particular interest is to "slide down" the transference between the data base application and the access paths, toward their lower-level representations, for reasons of efficiency (think, for example, of setting inversion records to point to data file records). Also, users of adequate degree of expertise may gain interfaces at various points in the architecture.

4. Ongoing Work

Since all kinds of modules discussed here are specified using the same formalism, the correctness of the architecture can be verified as it is developed. We are currently investigating appropriate methodologies for this.

It is especially important to verify that the architecture preserves the behavior of the data base application initially specified. This involves, among other problems, proving the faithfulness of representations and the sufficiency of interfaces to jointly handle the entire data base application. We would also like to determine how the execution of operations at each interface affects or is affected by operations executed at the other interfaces.

References

[1] ANSI/X3/SPARC interim report - FDT bulletin of ACM/SIGMOD 7,2 (1975).

[2] M.L. Brodie - Data abstraction, databases, and conceptual modelling : an annotated bibliography - NBS special publication 500-59 (1980).

[3] R.M. Burstall and J.A. Goguen - CAT, a system for the structured elaboration of correct programs from structured specifications - working draft , UCLA and SRI (1979).

[4] P.P. Chen - The entity-relationship model: towards a unified view of data - ACM/TODS 1,1 (1976) - 9-36.

[5] H. Ehrig, H.J. Kreowski and H.J. Weber - Algebraic specification schemas for data base systems - Proc. VLDB-Berlin (1978) 427-440.

[6] A.L. Furtado and P.A.S. Veloso - Procedural specifications and implementations for abstract data types - ACM/SIGPLAN Notices - to appear.

[7] J.A. Goguen, J.W. Thatcher and E.G. Wagner - An initial algebra approach to the specification, correctness, and implementation of abstract data types - in : Current Trends in Programming Methodology - v. IV - R.T. Yeh (ed.) Prentice Hall (1978) - 80-149.

[8] J.V. Guttag, E. Horowitz and D.R. Musser - The design of data type specifications - in : Current Trends in Programming Methodology - v. IV - R.T. Yeh (ed.) Prentice Hall (1978) - 60-79.

[9] J.V. Guttag, E. Horowitz and D.R. Musser - Abstract data types and software validation - ACM/Communications 21,12 (1978) - 1048-1064.

[10] G. Huet and D.C. Oppen - Equations and rewrite rules, a survey - Computer Science Department, Stanford University, Report No. STAN-CS-80-785 - (1980).

[11] C.B. Jones - Software development : a rigorous approach - Prentice Hall (1980).

[12] P.C. Lockemann, H.C. Mayr and K.R. Dittrich - A pragmatic approach to the algebraic specification of software modules - Universitat Karlsruhe, Fakultat fur Informatik, Interner Bericht Nr. 1/79 (1979).

[13] T.S.E. Maibaum and C.J. Lucena - Higher order data types - International Journal of Computer and Information Sciences 9,1 (1980) - 31-53.

[14] T.H.C. Pequeno and P.A.S. Veloso - Do not write more axioms than you have to - Proc. International Computer Symposium, Nankang (1978) - 488-498.

[15] C.S. dos Santos, E.J. Neuhold and A.L. Furtado - A data type approach to the entity-relationship model - in : Entity-relationship approach to systems analysis and design - P.P. Chen (ed.) - North Holland (1980) - 103-119.

[16] K.C. Sevcik and A.L. Furtado - Complete and compatible sets of update operations - Proc. International Conference on Database Management Systems - Milano (1978) 247-260.

[17] T.W. Tompa - A practical example of the specification of abstract data types - Acta Informatica 13,3 (1980) - 205-224.

[18] H.K.T. Wong and J. Mylopoulos - Two views of data semantics : a survey of data models in artificial intelligence and database management - INFOR 15,3 (1977) - 344-382.

```
type agdb

  op initag( ):agdb
     ⇒ INITAG
  endop

  op apply(x:person,s:agdb):agdb
     var z:person,w:company,n:natural,s1:agdb
     ~(~ iscandidate(x,s) ∧~ worksfor(x,?,s)) ⇒ s
     match s
       HIRE[z|w|s1] ⇒ HIRE[z|w|apply(x,s1)]
       SUBSCRIBE[w|n|s1] ⇒ SUBSCRIBE[w|n|apply(x,s1)]
       APPLY[z|s1] ⇒ if x > z then APPLY[z|apply(x,s1)]
                     else APPLY[x|s]
       otherwise ⇒ APPLY[x|s]
     endmatch
  endop

  op subscribe(y:company,m:natural,s:agdb):agdb
     var z:person,w:company,n:natural,s1:agdb
     m = 0 ⇒ s
     match s
       HIRE[z|w|s1] ⇒ HIRE[z|w|subscribe(y,m,s1)]
       SUBSCRIBE[w|n|s1] ⇒ if y = w then SUBSCRIBE[y|n+m|s1]
                           else if y > w then SUBSCRIBE[w|n|subscribe(y,m,s1)]
                                else SUBSCRIBE[y|m|s]
       otherwise ⇒ SUBSCRIBE[y|m|s]
     endmatch
  endop

  op hire(x:person,y:company,s:agdb):agdb
     var z:person,w:company,s1:agdb
     ~(iscandidate(x,s) ∧ haspositions(y,s) > 0) ⇒ s
     match s
       HIRE[z|w|s1] ⇒ if x > z then HIRE[z|w|hire(x,y,s1)]
                      else HIRE[x|y|s]
       otherwise ⇒ HIRE[x|y|s]
     endmatch
  endop

  op fire(x:person,y:company,s:agdb):agdb
     var z:person,w:company,s1:agdb
     ~ worksfor(x,y,s) ⇒ s
     match s
       HIRE[z|w|s1] ⇒ if x = z then s1
                      else HIRE[z|w|fire(x,y,s1)]
     endmatch
  endop
```

```
op iscandidate(x:person,s:agdb):logical
   var z:person,s1:agdb
   match s
     HIRE[z|?|s1] ⇒ if x = z then F
                    else iscandidate(x,s1)
     SUBSCRIBE[?|?|s1] ⇒ iscandidate(x,s1)
     APPLY[z|s1] ⇒ if x = z then T
                   else if x > z then iscandidate(x,s1)
                        else F

     otherwise ⇒ F
   endmatch
endop

op haspositions(y:company,s:agdb):natural
   var w:company,n:natural,s1:agdb
   match s
     HIRE[?|w|s1] ⇒ if y = w then haspositions(y,s1) - 1
                    else haspositions(y,s1)
     SUBSCRIBE[w|n|s1] ⇒ if y = w then n
                         else if y>w then haspositions(y,s1)
                              else 0

     otherwise ⇒ 0
   endmatch
endop

op worksfor(x:person,y:company,s:agdb):logical
   var z:person,w:company,s1:agdb
   match s
     HIRE[z|w|s1] ⇒ if x.y = z.w then T
                    else if x > z then worksfor(x,y,s1)
                         else F

     otherwise ⇒ F
   endmatch
endop

endtype
```

FIG. 1

```
transference agdb to set

  ...............

  op sempcomp(y:company,s:agdb):set
     ⇒ buildsetz(y,s,empty( ))
  endop

  hidden op buildsetz(y:company,s:agdb,z:set):set
     var x:person
     ifanotheremp(?x,y,s,z) ⇒ buildsetz(y,s,insert(x,z))
     ⇒ z
  endop

  hidden op ifanotheremp(x:person,y:company,s:agdb,z:set):logical
     ⇒ worksfor(x,y,s) ∧∼ has(x,z)
  endop

  ..................

endtransference
```

FIG. 2

```
interface of C

  ............

  op C-hire(x:person,s:agdb):agdb
     haspositions(C,s) > 10 ⇒ hire(x,C,apply(x,s))
     ⇒ hire(x,C,s)
  endop

  ............

endinterface
```

FIG. 3

```
type erm

  op φ( ):erm
     ⇒ $
  endop

  op cr(x:ent,t:eset,s:erm):erm
     var y:ent,z:ent,u:eset,a:attr,i:val,r:rel,s1:erm
     exs(x,t,s) ⇒ s
     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|cr(x,t,s1)]
       MOD[y|a|i|s1] ⇒ MOD[y|a|i|cr(x,t,s1)]
       CR[y|u|s1] ⇒ if x.t > y.u then CR[y|u|cr(x,t,s1)]
                    else CR[x|t|s]
       otherwise CR[x|t|s]
     endmatch
  endop

  op mod(x:ent,a:attr,i:(val,{*}),s:erm):erm
     var y:ent,z:ent,b:attr,j:val,r:rel,s1:erm
     ∿ (exs(x,?,s) ∧∿ hv(x,a,i,s)) ⇒ s
     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|mod(x,a,i,s1)]
       MOD[y|b|j|s1] ⇒ if x.a = y.b then
                          if i = * then s1
                          else MOD[x|a|i|s1]
                       else if x.a > y.b then
                          MOD[y|b|j|mod(x,a,i,s1)]
                            else MOD[x|a|i|s]
       otherwise ⇒ MOD[x|a|i|s]
     endmatch
  endop

  op lk(x:ent,y:ent,r:rel,s:erm):erm
     var z:ent,w:ent,q:rel,s1:erm
     ∿(exs(x,?,s) ∧ exs(y,?,s) ∧∿ isr(x,y,r,s)) ⇒ s
     match s
       LK[z|w|q|s1] ⇒ if x.y.r > z.w.q then
                         LK[z|w|q|lk(x,y,r,s1)]
                      else LK[x|y|r|s]
       otherwise ⇒ LK[x|y|r|s]
     endmatch
  endop

  op del(x:ent,t:eset,s:erm):erm
     var y:ent,z:ent,u:eset,v:eset,a:attr,i:val,r:rel,s1:erm
     ∿(exs(x,t,s) ∧ (inothereset(x,?v,t,s) ∨
                     (∿ isr(x,?,?,s) ∧∿ isr(?,x,?,s) ∧∿ hv(x,?,?,s)))) ⇒ s

     match s
       LK[y|z|r|s1] ⇒ LK[y|z|r|del(x,t,s1)]
       MOD[y|a|i|s1] ⇒ MOD[y|a|i|del(x,t,s1)]
       CR[y|u|s1] ⇒ if x.t = y.u then s1
                    else CR[y|u|del(x,t,s1)]
     endmatch
  endop
```

```
op ulk(x:ent,y:ent,r:rel,s:erm):erm
   var z:ent,w:ent,q:rel,sl:erm
   ∿ isr(x,y,r,s) ⇒ s
   match s
     LK[z|w|q|sl] ⇒ if x.y.r = z.w.q then sl
                    else LK[z|w|q|ulk(x,y,r,sl)]
   endmatch
endop

op exs(x:ent,t:eset,s:erm):logical
   var y:ent,z:ent,v:eset,a:attr,i:val,r:rel,sl:erm
   match s
     LK[y|z|r|sl] ⇒ exs(x,t,sl)
     MOD[y|a|i|sl] ⇒ exs(x,t,sl)
     CR[y|v|sl] ⇒ if x.t = y.v then T
                  else if x.t > y.v then exs(x,t,sl)
                       else F
     otherwise ⇒ F
   endmatch
endop

op hv(x:ent,a:attr,i:val,s:erm):logical
   var y:ent,z:ent,b:attr,j:val,r:rel,sl:erm
   match s
     LK[y|z|r|sl] ⇒ hv(x,a,i,sl)
     MOD[y|b|j|sl] ⇒ if x.a.i = y.b.j then T
                     else if x.a > y.b then hv(x,a,i,sl)
                          else F
     otherwise ⇒ F
   endmatch
endop

op isr(x:ent,y:ent,r:rel,s:erm):logical
   var z:ent,w:ent,q:rel,sl:erm
   match s
     LK[z|w|q|sl] ⇒ if x.y.r = z.w.q then T
                    else if x.y.r > z.w.q then isr(x,y,r,sl)
                         else F
     otherwise ⇒ F
   endmatch
endop

hidden op inothereset(x:ent,v:eset,t:eset,s:erm):logical
   ⇒ exs(x,v,s) ∧ v ≠ t
endop

endtype
```

FIG. 4

```
representation agdb by erm

   ............

   op hire(x:person,y:company,s:agdb):agdb
      var s1:erm
      ~(iscandidate(x,s) ∧ haspositions(y,s) > 0) ⇒ s
      match s
        REPAG[s1] ⇒ REPAG[lk(x,y,WORKS,cr(x,EMP,del(x,CAND,s1)))]
      endmatch
   endop

   ............

   op haspositions(y:company,s:agdb):natural
      var x:ent,n:natural,s1:erm
      match s
        REPAG[s1] ⇒ if isr(?x,y,WORKS,s1) then
                        haspositions(y,REPAG[ulk(x,y,WORKS,s)]) - 1
                    else if hv(y,NPOS,?n,s1) then n
                         else 0
      endmatch
   endop

   ............

endrepresentation
```

FIG. 5

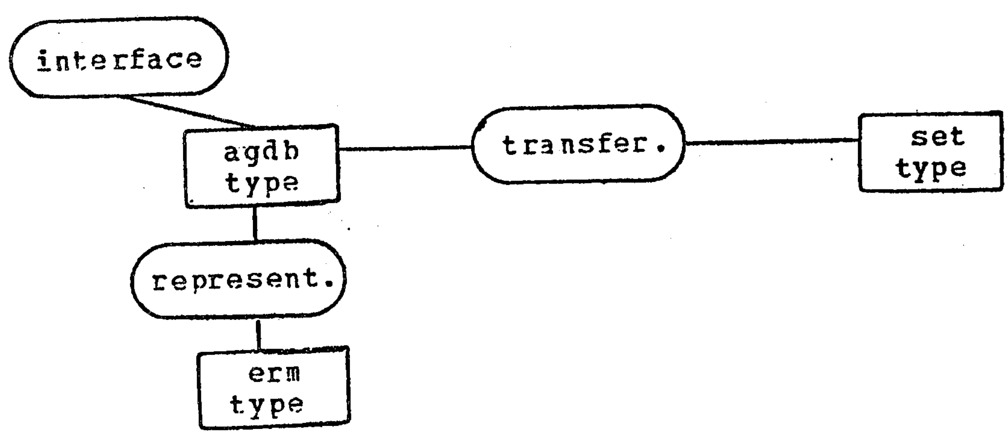

FIG. 6