Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions

V. Linnemann, K. Küspert, P.Dadam, P.Pistor,
R. Erbe, A. Kemper*, N. Südkamp, G. Walch, M. Wallrath*

IBM Scientific Center Heidelberg, Tiergartenstrasse 15
D-6900 Heidelberg, West Germany

* University of Karlsruhe, Fakultät für Informatik
D-7500 Karlsruhe, West Germany

Abstract

Current query languages for relational databases usually are fixed, i.e. they provide only a fixed set of data types and operations. It is usually not possible to extend this set by user defined data types or functions. This is a major drawback especially in advanced applications like engineering applications or office automation. In these areas special data types and special functions are needed quite frequently, e.g. a data type for matrices and a function for matrix multiplication. Since matrices and matrix multiplication are not provided in conventional query languages, the user has to model matrices by low level constructs as, for example, byte strings, and to write a rather cumbersome application program in a conventional programming language for interpreting these byte strings as matrices and for multiplying them. Another example of a missing function is even as simple as the square root function. Therefore, a mechanism is needed that allows the user to define his own data types and functions and add them somehow to the DBMS such that they can be used within the query language in the same way as a normal built-in function on basic data types. This paper describes an extension mechanism for data types and functions that has been implemented at the IBM Scientific Center in Heidelberg. The mechanism is based upon HDBL, an SQL based query language for complex objects. The functions themselves are written in a conventional programming language, in our case PASCAL, thus allowing to formulate general algorithms. One important aspect is the interface between the data types of the DBMS on the one side and the data types of the programming language on the other side. In our case, this mapping is more complicated than in other approaches because our type system supports complex objects directly and not via long strings as other authors do. Moreover, we use the PASCAL type system to a large extend in order to allow type checking at compile time.

1. Introduction

Current database management systems (DBMSs) and their database languages only offer a fixed set of data types and operations. Whenever the set of data types supported by the DBMS is insufficient for a given application, the system has to be "misused" in some way to handle the new type of data. Examples of that kind are large numerical vectors and matrices, long texts, geometrical data, image data, etc. In such cases the DBMS is used just as a "byte container". As a consequence, search predicates on the contents of these fields are usually not supported. The manipulation of these attribute values by the DBMS's DML can only be performed in a very rudimentary way. Moreover, a high dependency between the physical data representation and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

The work described in this paper was done within the R^2D^2 (A Relational Robotics Database System with Extensible Data Types) project. R^2D^2 is a cooperation project (started in 1986) between the IBM Scientific Center Heidelberg and the University of Karlsruhe, Fakultät für Informatik

the application programs is again re-established. - To lessen this kind of dependency was one of the major reasons why DBMSs have been developed.

Several ongoing research projects attempt to overcome limitations of current DBMSs by more powerful data models. Some of them support a richer set of basic data structures based on or influenced by nested relations or by variants of the entity-relationship model (/HL82, Da87, DaKü86, La84, RKB85, SS86, PaSc87, Hä87, AB84, Di86, VKC86/). Others try to solve the problem by staying with a more rigid basic data model but by supporting some kind of dynamic references. This is done in POSTGRES /RoSt87, St87, St86a, St86b, St84/ by introducing procedures as attribute values. These procedures consist, among others, of POSTQUEL statements, the database language of POSTGRES. Procedures as attribute values provide a powerful extension to the relational data model. In /St87/ it is shown how procedures as attribute values can be used to model complex objects.

Even with more powerful data structures offered, there will always remain cases, especially in novel application areas, where some new type of data cannot adequately be supported. Though some of the data structures offered by an advanced DBMS may be useful for efficiently storing the data, the search capabilities provided by the DBMS will usually be unsatisfactorily. Missing functions are not only a problem for "strange" data types. Assume for example the square root of a specific field or a specific group of fields. In a standard query language for a relational database, as for example SQL /Ch76, Ch81, IBM81/, one cannot express this problem because square root operations are usually not provided. The only way for the user is to forget about the query language and to write a rather cumbersome application program in a programming language. This is, of course, only possible if an interface to a programming language is provided.

One solution for this specific problem would be, of course, to ask the implementor of the DMBS to add the square root to the query language. But then the next user may ask for another function, e.g. matrix inversion. It obviously does not make sense to keep adding functions to the query language from the very beginning because there will always be applications which need other functions. What is really needed is a mechanism that allows the user to specify a new function and to provide appropriate interfaces to the query language. That is, to make the DBMS itself extensible by user defined data types and operations.

Currently, the area of extended data base technology is quite heavily investigated. Some of the work reported in the literature shall be reviewed shortly.

The 'PETERLEE RELATIONAL TEST VEHICLE' (PRTV /To76/), which is known as one of the first running prototypes of a relational DBMS, had already a simple mechanism for so-called 'user extensions': The user could provide his own procedures (written in PL/1) which could then be used in query statements and called by the DBMS at run time. Since PRTV tables were always in first normal form (1NF), complex (hierarchical) data structures as procedure input and output could not be processed.

Galileo /ACO85/ is a strongly-typed, interactive conceptual language for database applications designed, among others, to support the abstraction mechanisms of modern programming languages. The main contributions of Galileo are a flexible type system, the inclusion of type hierarchies and a mechanism to support abstract data types.

TAXIS /MBW80/ is a language for the design of interactive information systems. It offers, among others, database management facilities which are integrated into a single language through the concepts of class, property, and the IS-A relationship.

In addition to procedures consisting of POSTQUEL statements, POSTGRES /RoSt87, St87, St86a, St86b, St84/ also supports procedures written in a conventional programming language as, for example LISP or C. Moreover, the concept of abstract data types is supported by POSTGRES, but only on a rather low level as far as the representation of an abstract data type is concerned. The representation is an unstructured storage area. Only the length of the area is given, i.e. there is no strong typing as far as the representation of an abstract data type is concerned. This is also the method for passing parameters to functions written in LISP or C /St86a/.

PROBE /Da87, GO87/ distinguishes between *entities* and *functions*. Access to the attribute values of an entity is only provided by invoking the corresponding function. Functions can be system provided functions or user defined functions.

The STARBURST project /Sch86, LMP87/ is investigating, among others, how to design the DBMS architecture such that storage alternatives for relations and "foreign" indexes can be supported.

GENESIS /Bat86/ and EXODUS /Ca86/ are, in essence, software engineering tools for configurating a DBMS according to a given specification. GENESIS, for example, relies on database components whose interfaces have been standardized in such a way that they become exchangeable. One goal of EXODUS is to provide kernel DBMS facilities and software tools for the semi-automatic generation of application-specific DBMSs. Under the assumption that in the future there will exist large libraries of application area oriented data types and respective functions which can be optionally added to a data-

base kernel (customization), tools like GENESIS or EXODUS will be very helpful if not even mandatory to configure these systems.

"Extensibility" of a DBMS has several aspects. One is, how to make new data types and functions available to the user. That is, how to reflect them in the query language and in the application program interface. Another aspect is how to implement these functions. That is, how to program them ("what is the reference basis?"), how to "plug" them into the system, and how to actually execute them at run-time. A third aspect is how to support also user defined indexes within the DBMS, how to evaluate them during query optimization and execution, and how to integrate them into the system's concurrency control and recovery mechanisms.

In the R^2D^2 project we are currently mainly concentrating on the first two issues. In /KLW87/ the concept of abstract data types on top of nested relations is described. Our paper describes the extensibility of the underlying DBMS by user defined data types and functions and how they are reflected in its query language. The functions themselves are written in a conventional programming language, in our case in PASCAL, to allow for general algorithms. The underlying DBMS is a further development of the *Advanced Information Management Prototype*, called AIM-P in the sequel for short. AIM-P is an experimental DBMS developed at the IBM Heidelberg Scientific Center since 1983 for application oriented research purposes in advanced application areas (cf. e.g. /DaKü86, KDG87, Lu84, Lu85, Pi87/). AIM-P has been extended according to R^2D^2's needs. The link between AIM-P's database language and a user defined function is provided by mapping the data model of AIM-P to appropriate PASCAL structures and vice versa. It should be noted that the approach is not restricted to PASCAL. Any programming language which supports static types could be used as well, for example MODULA /Wi83/.

The paper is organized as follows: Section 2 discusses possible alternatives for adding types and functions to a DBMS by concentrating on the alternatives: static types versus dynamic types. Moreover, the relationship between abstract data types and so called encapsulated types is discussed. Section 3 recalls some database language constructs which are necessary for understanding section 4 which in turn is the central part of the paper. It describes by examples the function extension mechanism we have implemented. Implementation details are discussed in section 5. Section 6 gives some conclusions and an outlook for future work.

2. Static Types Versus Dynamic Types, Abstract Data Types Versus Encapsulated Types

If one talks about types and functions, there are two main alternatives: The types may be static or dynamic, i.e. the types of the parameters and the result of a function may be statically known, or they may vary dynamically. In POSTGRES /RoSt87, St87, St86a, St86b/ , even the type of a tuple in a table may vary from tuple to tuple. This results from the fact that each function stored in an attribute value may produce a value of an arbitrary type. Opposed to a "normal" attribute, the structure (value type) is therefore not known prior to the access to the attribute value and to the execution of it (that is the function/procedure it contains). This approach provides a lot of flexibility. On the other hand, only dynamic type checking can be provided, i.e. type errors show up only at run time. To write an application program for processing tables with tuples of unpredictable types is rather difficult and error prone. Therefore, we think that for the "standard" user more secure mechanisms should be provided. Moreover, optimization is easier if types are known.

Improved security and efficiency can be achieved by binding functions to *static types*. On the database programming language side, probably the most significant contributions supporting static types were PASCAL/R /Schm77/ and Galileo /ACO85/. By using static types, the result type of a function can be determined respectively derived at function definition time. Thus it can be described in the catalog (cf. Sect. 5.1.2). By doing so, the data structures returned when executing a function are already known at compile time of the application program. Hence, there are no "surprises" at execution time.

For these reasons we have decided for R^2D^2 to bind the functions with respect to their parameters and return values to *static types*. Therefore, only type compatible attribute values, constants, and query expressions can be passed as actual parameter to these functions.

In R^2D^2 functions are not limited to basic data types like integer, real, string, etc. A function can be defined on any data structure supported by AIM-P; even a complete table as data type is allowed. Therefore more emphasis than in the flat table case had to be put on providing a reasonable basis for the implementation of these functions.

At this point some comments should be made on abstract data types. We feel that the database kernel should provide more than pure abstract data types. Binding of functions to only one type is too narrow because there are applications where a

function belongs to two or more types. Consider, for example, the problem of converting the value of one abstract data type to a value of another abstract data type. This conversion function belongs to both abstract data types. If only pure abstract data types are supported, the conversion function has to be added artificially to one of the two abstract data types. Therefore, we decided to directly support only the information hiding concept of abstract data types by introducing so called *encapsulated types*. The structure of encapsulated types is not known to the user, values of an encapsulated type can be accessed and changed only by appropriate functions. Functions can refer to several encapsulated types. Encapsulated types are similar to the concept of *hidden types* introduced by the programming language MODULA /Wi83/. The concept of abstract data types is, in a sense, a special case of encapsulated types because an abstract data type is an encapsulated type together with functions restricted to this type.

One of the main goals of R^2D^2 is to provide an environment where adding of new functions to the underlying DBMS should be possible without requiring much database specific knowledge. Especially, it should not require knowledge about internals of the underlying DBMS, especially the internal data representation. Every experienced application programmer should be able to program these functions. In order to make this a safe task, the functions should be implemented with programming language structures which represent the corresponding data model types as "naturally" as possible. This means that a tuple, for example, should be mapped to a record structure rather than to a byte string with offset pointers.

To understand our approach for solving the function implementation problem, a brief explanation of the underlying data model has to be given first.

3. Data Model and Language

The data model supported by the Advanced Information Management Prototype is an object-oriented generalization of Non-First-Normal-Form (NF^2) respectively "nested" relations. It has an SQL-like language interface, the **Heidelberg Data Base** Language (HDBL) /PT86, PA86, Pi87/.

The **object types** HDBL can deal with are:

set valued, list valued, tuple valued or atomic.

Atomic data types are: DATE, REAL, INTEGER, BOOLEAN, CHARACTER, STRING and SURROGATE. The elements or attributes of any object type (except for objects consisting just of an atomic value) can again be of any of the types listed above. That is, the attributes of a tuple valued object, for example, can be either atomic, or set valued, or list valued, or again tuple valued. Objects need not occur as elements of a table. A list of lists of REAL values (which is a two dimensional matrix) can occur as element in another list or set or as attribute value within a tuple or as a single standing object (having an object name). Figure 1.a shows a graphical representation of this data model; both the 1NF data model and the pure NF^2 data model are special cases of this more general data model.

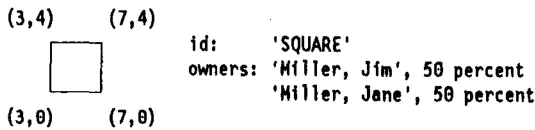
As we will use HDBL statements later on to show the embedding of user defined functions and types, we give here a brief introduction into this language. A comprehensive treatment of this subject can be found in /PT86, PA86, Pi87/.

The following example shows a CREATE statement and some simple queries. The example will later on also serve as reference basis for the discussion of user defined data types and functions. To make explanation not unnecessarily complicated we have selected a rather simple structure. It should be clear, however, that HDBL can deal with much more complex structures and operations on those (projection, selection, join) as well.

As an example we use a part of a geographic information system which allows to store information about specific properties. Each property is defined by the boundaries which are given by a list of points. We can create a corresponding table in HDBL as follows:

```
CREATE properties
    { [ id: string(10),
        owners: { [ name: string(30),
                    share: real ] },
        points:   < [ x_c: real,
                      y_c: real ] > ] }
END
```

Sets are indicated by curly brackets ({...}), tuples by square brackets ([...]), and lists (ordered sets) by sharp brackets (<...>) /ALPS88/. Thus, the properties example represents a set of tuples. Each tuple has three attributes: The first one, called 'id', is an identifier of the property. 'id' is a string, i.e. a flat attribute in the conventional sense. The second one, 'owners', represents all the owners of the property. 'owners' is a set of tuples. Each tuple contains the name of an owner and the percentage of the ownership. The third one, called 'points', represents the boundary of the property. It is a list of tuples. Each list element represents a limiting point. A list element is a tuple containing the x and y coordinates of the limiting point. For example, the property depicted by the picture



can be modelled by the following properties table entry:

id	{ owners }		< points >	
	name	share	x_c	y_c
SQUARE	Miller, Jim Miller, Jane	50.0 50.0	3.0 7.0 7.0 3.0	4.0 4.0 0.0 0.0

In HDBL, the information about properties '1234' and '5678' can be retrieved by

```
SELECT p  FROM p IN properties
WHERE p.id = '1234' OR p.id = '5678'
```

If one is only interested in the *owners* of the specified properties, one can formulate the following projection:

```
SELECT [p.owners] FROM p IN properties
WHERE p.id = '1234' OR p.id = '5678'
```

If one is interested in all properties such that a specific point occurs in the limiting points, one can express this in HDBL as follows:

```
SELECT p FROM p IN properties
WHERE EXISTS (point IN p.points):
      point.x_c = 13.7 AND point.y_c = 39.8
```

In addition to queries, HDBL provides operations for changing tables (ASSIGN, INSERT and DELETE). ASSIGN assigns a new value to a specific field whereas INSERT inserts one or several elements into an existing set or list. DELETE deletes one element or a whole set or list. Assume, for example, that a property share is split into two parts. This can be expressed by the following two HDBL statements:

```
ASSIGN owner.percentage * 0.5
TO owner.percentage
FROM owner IN p.owners, p IN properties
WHERE  p.id = 'XYZ' AND owner.name = 'Mr. X'

INSERT { [ name: 'Mr. Y',
           percentage: owner.percentage ] }
INTO p.owners
FROM p IN properties, owner IN p.owners
WHERE  p.id = 'XYZ' AND owner.name = 'Mr. X'
```

4. User Defined Types and Functions

Although being quite powerful, HDBL does not allow certain queries. One group of queries involves, among others, the computation of the transitive closure of a relation. This problem could be solved by introducing recursive queries over nested relations /Li87/. Another group of queries involves the computation of arbitrary functions involving, among others, mathematical expressions. Assume, for example, the query:

"Find all properties such that the length of the boundary is larger than a certain value."

This query cannot be expressed in current HDBL because it involves looping over all limiting points and the computation of the square root. One solution to solve problems of that kind is to use the application program interface /ESW87, EW87/ for fetching the objects of interest and to perform the computation itself in the application program.

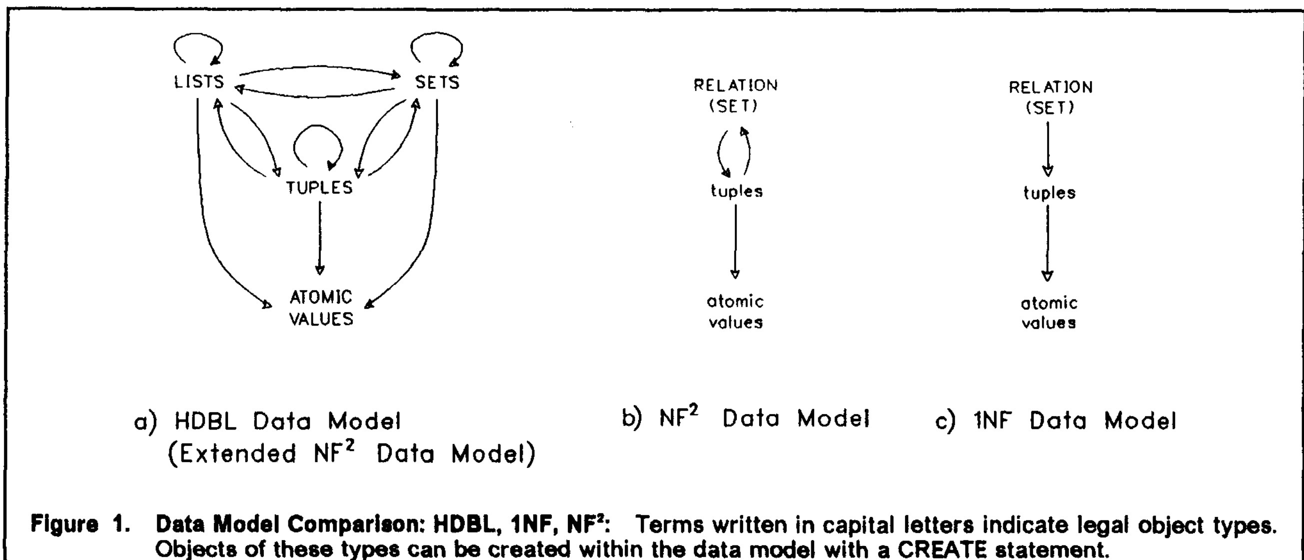


Figure 1. Data Model Comparison: HDBL, 1NF, NF²: Terms written in capital letters indicate legal object types. Objects of these types can be created within the data model with a CREATE statement.

Especially for computations which are needed frequenty, especially if they are needed in various applications (think, for example, just a square root or standard deviation function is missing) this approach is too cumbersome. Hence a mechanism should be provided to make the DBMS itself *extensible* by user defined functions such that they become part of the DBMS's query language. This section describes how this has been achieved in the Advanced Information Management Prototype

For the user, the most obvious solution to the query: Find all properties such that the length of the boundary is larger than a certain value, would be to define a function 'get_length' which computes the length of a boundary and then use this function in the following HDBL statement:

```
SELECT p FROM p IN properties
WHERE get_length(p.points) > 123456.7
```

It should be possible to program 'get_length' in a programming language like PASCAL. One important point has to be solved for that end: The world of PASCAL types has to be connected to the world of HDBL types, because PASCAL functions like 'get_length' require parameters of PASCAL type. In our approach, this is accomplished as follows: A special DECLARE TYPE statement is added to HDBL which allows the user to define types which can be used in CREATE statements or within other DECLARE TYPE statements. Once a type has been declared, the system will generate corresponding PASCAL representations (type declarations) for this type. For example, the statement

```
DECLARE TYPE point
            [ x_c: real,
              y_c: real ]
END
```

defines a type 'point' as a tuple with an x coordinate and a y coordinate. The translation of the DECLARE TYPE statement results in the generation of a corresponding PASCAL type declaration as follows (cf. Sect. 5.3 for the details):

```
TYPE point   = RECORD
                  x_c: real;
                  y_c: real
               END;
```

By the statement

```
DECLARE TYPE boundary < point > END
```

a type boundary is defined to be a list of points. For this HDBL type the following PASCAL types would be generated:

```
TYPE boundary$R = ARRAY [ 1..65535 ] OF point;
     boundary   = RECORD
                     ACT_ELEM: 0..65535;
                     ALO_ELEM: 0..65535;
                     val     : ↑boundary$R;
                  END;
```

These types need some explanations: Since PASCAL like many other programming languages does not support dynamic arrays, "special solutions" have to be used to overcome the problems of representing variable long lists or sets. In our example, a default limit of 65535 is used, since no limit was given in the declare statement. The components 'ACT_ELEM', 'ALO_ELEM' and 'val' simulate a dynamic array. In the val component, the list elements are stored. 'ACT_ELEM' indicates the current length of the list. 'ALO_ELEM' is needed for storage allocation. This is described in more detail in Sect. 5.3.3. With these types, the 'properties' table can now be defined as

```
CREATE properties
   { [ id:         string(10),
       owners:     { [ name: string(30),
                       percentage: real ] },
       points:     boundary
     ] }
END
```

From the database point of view this statement is equivalent to the first CREATE statement, since the types are not declared to be encapsulated. If a type is declared to be encapsulated, the internals of a value of such a type are known only to the functions which have a value of such a type as a parameter. For example, by the declaration

```
DECLARE TYPE boundary < point > ENC END
```

'boundary' is declared to be encapsulated, i.e. the elements of a boundary list cannot be accessed directly but only by functions which have a 'boundary' as a parameter.

After having declared the necessary types, we can introduce our function 'get_length'. This is done in two steps: First, 'get_length' is made known to the database system by the statement

```
DECLARE FUNCTION get_length(b: boundary): real
```

In a second step, the body of the function is written in PASCAL by using an auxiliary function 'line_length':

```
FUNCTION line_length(p1,p2: point): real;
VAR x,y: real;
BEGIN
 x := p2.x_c - p1.x_c;  y := p2.y_c - p1.y_c;
 line_length := SQRT( x*x + y*y );
END;

FUNCTION get_length(b: boundary): real;
VAR len: real; i: integer;
BEGIN
 WITH b DO
 BEGIN
  IF ACT_ELEM <= 2
  THEN BEGIN
       len := 0.0;
       error_exit(....) /* WRONG DATA: boundary must
                           have at least 3 points */
       END
  ELSE BEGIN
       len := line_length(val↑[ACT_ELEM], val↑[1]);

       FOR i:=1 TO ACT_ELEM-1 DO
         len := len + line_length(val↑[i], val↑[i+1])
       END
 END;
 get_length := len
END;
```

This PASCAL program is now compiled by the PASCAL compiler and added to the database software. More details are given in Sect. 5. 'get_length' can now be used within HDBL whereever a real value (= value type of the result) is allowed. One example was already given:

```
SELECT p FROM p IN properties
WHERE get_length(p.points) > 123456.7
```

Another example uses 'get_length' for the construction of an attribute value. It gives the properties table together with the lengths of the boundaries:

```
SELECT [ id : p.id, owners: p.owners, points: p.points,
         length: get_length(p.points) ]
FROM p IN properties
```

In the next example, we want to change our 'properties' table such that the name is split in first name and last name. For this example, we use another alternative for defining types, namely a DERIVE TYPE statement which derives a type from an existing table. We declare types for the owners as follows:

```
DECLARE TYPE new_owner
  [ first_name: string(30), last_name: string(30),
    percentage: real ]
END

DERIVE TYPE old_owner AS o
FROM o IN pr.owners, pr IN properties
```

The system generates the following PASCAL types:

```
TYPE new_owner = RECORD
                    first_name: string(30);
                    last_name:  string(30);
                    percentage: real
                 END;

TYPE old_owner = RECORD
                    name: string(30);
                    percentage: real
                 END;
```

A function for splitting names is made known to the system by

```
DECLARE FUNCTION name_split (old: old_owner): new_owner
```

The corresponding PASCAL implementation can be sketched as follows:

```
FUNCTION name_split (old: old_owner): new_owner;
VAR result: new_owner;
BEGIN
    result.last_name  := ...;
    result.first_name := ...;
    result.percentage := old.percentage;
    name_split := result
END;
```

Now we can create our new table:

```
CREATE new_properties
   { [ id: string(10),
       owners: { new_owner }
       points:     < [ x_c: real,
                       y_c: real ] >
     ] }
END
```

and convert the old table as follows:

```
INSERT
 SELECT [ id    : old_prop.id,
          owners: SELECT name_split(old_owner)
                  FROM old_owner IN old_prop.owners,
          points: old_prop.points ]
 FROM old_prop IN properties
INTO new_properties
```

5. Implementation of Types and Functions

In the following we will explain how user defined data types and functions have actually been implemented within the DBMS. We address database catalog extensions (Sect. 5.1), the run time management of user defined functions and type instances (Sect. 5.2), and - finally - the PASCAL data structures chosen to map the HDBL types into (Sect. 5.3).

5.1 Types and Functions in the Database Catalog

5.1.1 The Object Catalog of AIM-P

The database catalog of AIM-P is composed of

three parts: the object catalog, the type catalog, and the function catalog. The last two catalogs reflect the extensions of our system by types and functions, the first one was ***the*** catalog in the initial version of AIM-P.

The **object catalog** records descriptive information - meta data - about all database objects, i.e. - in the HDBL data model - about sets, lists, tuples, and scalars (cf. Figure 1).

The object catalog itself is again an HDBL object (in fact a table). Each tuple in that table contains some general information about the related database object as a whole ('**one_object_description**'), such as its external and internal name, its creator and creation date, etc. The most important part of the object catalog, however, is the attribute description, which is a list of tuples where each tuple ('**one_attribute_description**') describes exactly one attribute. Via the attribute description the catalog manager of the DBMS keeps also track of the parent-child relationships between attributes on different levels, i.e. the attribute description reflects the hierarchical structure within a database object of arbitrary size and complexity.

5.1.2 Catalog Extensions for Types and Functions

Information about types and functions is maintained in two separate tables, a **type catalog** and a **function catalog**.

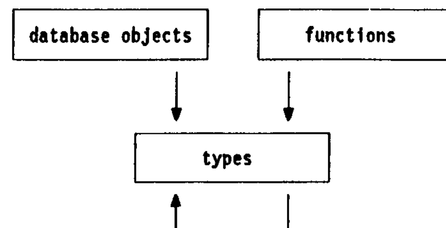
As explained in Sect. 4, any flat or nested HDBL structure (list, set, tuple, scalar) may be used in exactly the same way both in a type declaration (DECLARE TYPE ... or DERIVE TYPE ...) and in a database object declaration (CREATE ...). It is therefore not surprising that the structure of the **type catalog** is quite similar to that of the object catalog. Especially the attribute description is done in exactly the same way.

For each user defined data type the catalog manager records, again, the internal and external name, the creator and creation date, etc. The corresponding PASCAL data structure is automatically derived from the type declaration (see Sect. 5.3) and could be stored in the type catalog. Moreover, several '**use counts**' are provided to keep track of a type's usage in other types, in functions, and in database objects. We will come back to the semantics of these use counts at the end of this subsection.

In the **function catalog** the DBMS keeps track of the names and the interface descriptions (parameter names and types) of all user defined functions. Parameters of functions - as well as the function result - may either be of basic HDBL types (BOOLEAN, INTEGER, REAL, etc.) or of user defined types. The basic HDBL types are always implicitly known to the DBMS; the user defined types must have been explicitly declared before they can be used in a function definition.

Certain attributes in the function catalog, like 'function_result_type' and 'parameter_type', are used by the DBMS at DML parsing time to check the applicability of a given function in a DML statement. These type specifications for function input and output are also used to trigger implicit type conversions (as far as possible) if the actual and formal parameters do not fully match. A 'parameter_name' attribute is also provided to automatically generate the corresponding PASCAL function declaration to be used in a function implementation.

Our type and function concept had to be designed and implemented such that the DBMS is able to keep track of all interdependencies between types, functions, and database objects. These interdependencies can be expressed in the following diagram:



The figure reads as follows: Types may be used in the declaration of objects, functions, and (other) types. A consequence is that whenever a type shall be dropped the DBMS must check whether that type is still somewhere in use (in objects, functions, or types). We decided to permit type drop operations only if the type to be dropped is **not** (anymore) in use. An alternative would have been to implicitly drop also those objects, functions, and types where the type to be dropped is currently used by; we believe, however, that for the normal user the effects of such 'cascading drop operations' would be rather unpredictable and therefore extremely unsafe.

5.2 Handling of User Defined Functions and Type Instances at Run Time

5.2.1 Function Dispatcher

Obviously, user defined functions *cannot* be 'hardwired' since neither the function names nor the number and kind of parameters are known when the DBMS code is written. Extensibility means that new user defined functions can be brought into the DBMS at *any time*. The DBMS must be able to call these functions and to provide the proper set of

parameters. It would not be really satisfying if one had to change the DBMS source code every time when the user defines a new function. A mechanism is therefore required in the DBMS to call *any* function with *any* set of parameters. We call that mechanism the **function dispatcher**.

In the following the function dispatcher will be briefly explained by presenting the scenario in which a function call within a DML statement is actually processed by the DBMS:

When the DML statement is parsed the **parser** checks whether the specified function exists or not. This is done via a look up operation in the function catalog (Sect. 5.1.2). The parser also checks the applicability of the function in the given context of the DML statement, i.e. whether actual and formal parameters really match in number and type (if necessary type conversions are prepared).

At DML execution time the function must actually be called by the DML **evaluator** (run time evaluation part of the DML processor) and the required parameters must be passed. This function call is done *indirectly* via a call to the **function dispatcher**. The function dispatcher gets basically three input parameters, namely the function name, the number of parameters and an address vector with one address for each parameter.

The function dispatcher has some internal bookkeeping (**function address table**) to keep track of all function names and function start addresses. Whenever a new user defined function is brought into the system its function name and interface description are not only recorded in the function catalog, but also in the function address table.

The function dispatcher - written in assembly language - takes the given function name to look up the corresponding function start address and then executes this function. The address vector provided by the caller is passed to the function.

Once the user defined function has been executed the result is returned to the function dispatcher and - finally - to the DML evaluator where that result can be used e.g. as input for another function, for display at the screen, or for use in predicate evaluation, etc.

5.2.2 The Type Instance Loader/Unloader

In order to provide the actual (input) parameter set for user defined functions, AIM-P must be able to load (complex) type instances from the database into the respective Pascal structures. In our example in Sect. 4, where an instance of type 'boundary' is input for the function 'get_length', the required 'boundary' data - which are a list of 'point' tuples - must be read from the database and must thereby be transformed into the Pascal representation. This is both done by the **type instance loader**.

Moreover, AIM-P must also be able to unload (complex) type instances from a Pascal representation (back) to the database. An example was also given in Sect. 4: The result data of the 'name_split' function, which are of type 'new_owner', are finally inserted into the database. Data transformation from the Pascal representation to the database format and writing to the database are done by the **type instance unloader**.

The type instance loader and unloader shield the higher level DBMS components from details of type instance implementation, such as storage allocation, address and pointer representation, Pascal data layout, etc. For the AIM-P query processor there is functionally no difference between loading/unloading a simple scalar value (e.g. a 4 byte INTEGER) on the one hand and loading/unloading a large, complex type instance of size 4 MB on the other hand: Both is done via a single call to the type instance loader/unloader which is - in case of data loading - also responsible for storage space allocation.

The type instance loader and unloader are fully catalog driven: They have both the information about an object's database format (via the **object** catalog) and about its Pascal format (via the **type** catalog). They can thereby - on the fly - perform the necessary **conversions** between the database format and the respective Pascal format. These conversions are not restricted to atomic data occurrences (e.g. transformation of an INTEGER vector in the database to a REAL vector in Pascal and vice versa); other transformations such as LIST (n) ⇔ SET (m) etc. can be done as well.

5.3 Generation of PASCAL Data Structures for HDBL Types

When creating PASCAL data structures for HDBL types, three different *strategies* are conceivable:

User defined PASCAL structures: The user who defines a new type may specify whatever PASCAL data structure he would like to see for that type. To gain efficiency, to save storage space, and to write compact program code, the user can therefore tailor the PASCAL structure to the operations to be performed on that structure. A major disadvantage of this approach is, however, that no general mapping mechanism can be provided by the DBMS to transform data from the internal database format to the PASCAL format and vice versa. The user would have to implement these mapping routines himself, a task that is both cumbersome and extremely error-prone; incorrect mapping routines could even

destroy the database. We therefore did not follow that approach.

System defined PASCAL structures: One standard PASCAL equivalent is defined for each HDBL construct, i.e. for sets, lists, tuples, and scalars (cf. Figure 1). For any user defined type the DBMS can therefore automatically generate the PASCAL equivalent for that type. The mapping algorithm to transform data from the internal database format to the PASCAL format and vice versa is also fixed (catalog driven) and can thus be provided by the DBMS; no user driven transformation is required. An obvious disadvantage of system defined PASCAL structures is that the user has no means for optimization and customization; the user defined functions have to be coded on - and tailored to - the PASCAL data structures as they are provided by the DBMS.

User customized PASCAL structures: In principle, there are again standard PASCAL equivalents for HDBL types, as in case of system defined PASCAL structures. To a certain degree, however, the user may **customize** these PASCAL structures in a sense that the DBMS offers him a number of different PASCAL equivalents for each basic HDBL construct. A HDBL list, for instance, could be represented in PASCAL via an array, pointer array, or linked list implementation. Among these different choices the user may now select the most appropriate PASCAL data structures for his personal needs. Since customized PASCAL structures are not fully user defined but still DBMS controlled, general mapping routines can be provided by the DBMS and need not be manually coded by the user.

The approach of user customized PASCAL structures seems to be a fairly good compromise between user defined PASCAL structures on the one side (which require too much manual interaction) and system defined PASCAL structures on the other side (which are sometimes not appropriate for the implementation of specific algorithms). In our system, the last two approaches are supported. Further details on customization can be found in /KKLW87/.

Whenever a new type is defined (via DECLARE TYPE ... or DERIVE TYPE ...), an equivalent PASCAL declaration is created as well which can then be embedded into the source code of PASCAL functions working with that type. Some small examples for these PASCAL data structures have already been shown in Sect. 4 (see e.g. PASCAL types 'boundary' and 'point').

In the following subsections further examples for HDBL types and their PASCAL equivalents will be given together with a set of more *general rules* how to create system defined PASCAL data structures for given HDBL type definitions.

5.3.1 PASCAL Types for Basic HDBL Types and Previously Defined Types

For a type declaration

```
DECLARE TYPE  user_defined_name  any_type_name  END
```

the corresponding PASCAL type declaration is

```
TYPE user_defined_name = any_type_name;
```

For example, for the declaration

```
DECLARE TYPE  my_own_real  REAL  END
```

a PASCAL declaration with the contents

```
TYPE my_own_real = REAL;
```

is created.

5.3.2 PASCAL Types for HDBL Tuples

Assume now a HDBL *tuple type* definition with attribute names a_i and type names t_i:

```
DECLARE TYPE  user_defined_name
                 [ a1 : t1, . . ., an : tn ]
END
```

A PASCAL **record** type serves as the programming language construct to map a HDBL tuple type into:

```
TYPE user_defined_name = RECORD
                           a1 : t1; . . .; an : tn
                         END;
```

Instead of the type names t_i, the user may also specify any other DDL construct, e.g. another HDBL tuple, as the following example of *nested tuples* illustrates:

```
DECLARE TYPE  nested_tuples
                [ attribute_1 : REAL,
                  attribute_2 : [ attribute_3 : INTEGER,
                                  attribute_4 : CHAR    ] ]
END
```

Two PASCAL types are created for the two tuples in that HDBL type definition:

```
TYPE nested_tuples$attribute_2 =
        RECORD
          attribute_3 : INTEGER;
          attribute_4 : CHAR
        END;

     nested_tuples              =
        RECORD
          attribute_1 : REAL;
          attribute_2 : nested_tuples$attribute_2
        END;
```

Nested_tuples$attribute_2 is a system generated PASCAL type name.

5.3.3 PASCAL Types for HDBL Sets and Lists

For any HDBL set/list type definition, e.g.

```
DECLARE TYPE user_defined_name <n FIX element_type> END
DECLARE TYPE user_defined_name <n     element_type> END
DECLARE TYPE user_defined_name {n     element_type} END
```

where 'element_type' is the name of the element type and 'n' is the maximal (variable length) or actual (fixed length) number of elements, the PASCAL representation looks as follows:

```
TYPE user_defined_name$R = ARRAY [1 .. n] OF element_type;

     user_defined_name    =
       RECORD
         ACT_ELEM        : 0 .. n;
         ALO_ELEM        : 0 .. n;
         VAL             : ↑user_defined_name$R
       END;
```

ACT_ELEM gives the actual number of elements; ALO_ELEM gives the number of elements for which storage space has been allocated (ALO_ELEM ≥ ACT_ELEM). ALO_ELEM has been introduced since - in order to save space in main memory - one does not always want to allocate the **array** in its **maximal** length (n) which might be rather large (see also Sect. 4). In programming languages which directly support arrays of **variable** length, this construct could be simplified.

Instead of the type name 'element_type', the user may again specify any other DDL construct, e.g. another HDBL set or a tuple, thus defining sets of sets, sets of tuples, etc. without having to perform explicit type declarations for the lower level sets and tuples.

6. Status and Conclusions

In this paper we have described a mechanism for adding user defined data types and functions to a DBMS. We have outlined how functions are reflected in the query language, how they are to be implemented, and how they are executed at runtime. Moreover, we have described the system extensions performed in order to support these tasks. Though described for the Advanced Information Management Prototype, the solution is generally applicable. At the time being, only functions written in a programming language are supported. We have therefore concentrated on those in this paper. We plan, however, to support functions written in HDBL as well.

Because the functions are compiled (machine code) they are nearly as efficient as comparable standard built-in functions though some extra overhead caused by in-core data movement and data conversions has clearly to be paid for supplying the functions with their parameter values, and to put their results back into the DBMS's internal representation.

Supporting *compiled* instead of *interpreted* functions certainly increases the risk that a malfunction of a user provided function may cause the DBMS to stop. This risk could be avoided by putting the code of the function into a separate address space, a solution which has also been suggested for POSTGRES. This, however, would cause some additional performance penalty (task switch). For the time being, we execute both user defined functions and normal DBMS code within the same address space. As rather conventional data structures are provided to program these functions (no "trick programming" is required) and as dynamic storage allocation and de-allocation is done via dedicated allocation routines /KKLW87/, this risk seems to be tolerable.

Acknowledgement

The authors would like to thank A. Blaser, manager of the IBM Scientific Center in Heidelberg, for the continuous support of the Advanced Information Management Project

References

AB84 S.Abiteboul, N.Bidoit: Non First Normal Form Relations: An Algebra Allowing Data Restructuring. Rapports de Recherche No 347, Institut de Recherche en Informatique et en Automatique, Rocquencourt, France, Nov. 1984

ACO85 A.Albano, L.Cardelli, R.Orsini: Galileo: A Strongly-Typed, Interactive Conceptual Language, ACM Transactions on Database Systems, Vol. 10, No. 2, June 1985, pp. 230-260

ALPS88 F.Andersen, V.Linnemann, P.Pistor, N.Südkamp: Advanced Information Management Prototype: User Manual for the Online Interface of the Heidelberg Data Base Language (HDBL) Prototype Implementation, Release 2.0, Jan. 1988, IBM Scientific Center Heidelberg TN 86.01

Bat86 D.S.Batory et al.: GENESIS: A Reconfigurable Database Management System. Dept. of Comp. Science, University of Texas at Austin, TR-86-07, March 1986

Ca86 M.J.Carey, D.J.DeWitt, D.Frank, G.Graefe, M. Muralikrishna , J.E.Richardson, E.J.Shekita: The Architecture of the EXODUS Extensible DBMS, Proc. 1986 IEEE Intern. Workshop on Object Oriented Database Systems, Pacific Grove, pp. 52-65

Ch76 D.D.Chamberlin et al.: SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control, IBM Journ. Res. Devel. 20 (1976), pp. 560-575

Ch81 D.D.Chamberlin et al.: Support for Repetitive Transactions and Ad Hoc Queries in System R. ACM TODS, Vol. 6, No.1, March 1981, pp. 70-94

Da87 U. Dayal, F.Manola, A.Buchman, U. Chakravarthy, D.Goldhirsch, S.Heiler, J.Orenstein, A. Rosenthal : Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them, Informatik Fachberichte 136, Springer-Verlag 1987, pp. 17-37

DaKü86 P.Dadam, K.Küspert, F.Andersen, H.Blanken, R.Erbe, J.Günauer, V.Lum, P.Pistor, G.Walch: A

DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, Proc. ACM SIGMOD Conf., Washington,D.C., 1986, pp. 356-367

Di86 K.R. Dittrich: Object Oriented Database Systems: The Notion and the Issues, Proc. 1986 IEEE International Workshop on Object Oriented Database Systems, Pacific Grove, pp. 2-6

ESW87 R. Erbe, N. Südkamp, G. Walch: An Application Program Interface For A Complex Object Database, IBM Heidelberg Scientific Center Tech. Report TR 87.10.008, to appear in Proceedings of the 3rd Intern. Conference on Data and Knowledge Bases, Jerusalem, 1988

EW87 R. Erbe, G. Walch: An Application Program Interface for an NF^2 Database Language or How to Transfer Complex Object Data Into an Application Program. IBM Heidelberg Scientific Center, Tech. Rep. TR 87.04.003, April 1987

GO87 D.Goldhirsch, J.A.Orenstein: Extensibility in the PROBE Database System. Data Engineering, Vol. 10, No. 2, June 1987, pp. 24-31

Hä87 T.Härder, K. Meyer-Wegner, B.Mitschang, A. Sikeler PRIMA: a DBMS Prototype Supporting Engineering Applications. Proc. VLDB, Brighton, U.K., September 1987, pp. 433-442

HL82 R.L.Haskin, R.A.Lorie: On Extending the Functions of a Relational Database System. Proc. SIGMOD 82, Orlando, June 1982, pp. 207-212

IBM81 SQL/Data System, Concepts and Facilities, IBM Corporation, GH 24-5013, Jan. 1981

IBM85 PASCAL/VS Language Reference Manual, IBM Corporation, Program Number: 5796-PNQ, 1985

KDG87 K.Küspert, P.Dadam, J.Günauer: Cooperative Object Buffer Management in the Advanced Information Management Prototype. Proc. VLDB, Brighton, U.K., Sept.1987, pp. 483-492

KKLW87 A. Kemper, K. Küspert, V. Linnemann, M. Wallrath: Pascal Structures for HDBL Types: Layout, Naming Conventions, Storage Allocation, and Usage in Functions. IBM Heidelberg Scientific Center Tech. Note TN 87.05, Oct. 1987

KLW87 A. Kemper, P.C. Lockemann, M.Wallrath: An Object-Oriented Database System for Engineering Applications. Proc. ACM SIGMOD Conf., San Francisco, May 1987, pp. 299-311

La84 W.Lamersdorf: Recursive Data Models for Non-Conventional Database Applications, Proc. First Intern. IEEE Conference on Data Engineering, Los Angeles, 1984, pp. 143-150

Li87 V.Linnemann: Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach, Proc. Third IEEE Conference on Data Engineering, Los Angeles, 1987, pp. 591-598

LMP87 B.Lindsay, J.McPherson, H.Pirahesh: A Data Management Extension Architecture, Proc. SIGMOD 1987, San Francisco, May 1987, pp. 220-227

Lu84 V.Lum, P.Dadam, R.Erbe, J.Günauer, P.Pistor, G.Walch, H.-D.Werner, J.Woodfill: Designing DBMS Support for the Temporal Dimension. Proc. SIGMOD '84, Boston, Mass., June 1984, pp. 115-130

Lu85 V.Lum, P.Dadam, R.Erbe, J.Günauer, P.Pistor, G.Walch, H.-D.Werner, J.Woodfill: Design of an Integrated DBMS to Support Advanced Applications. Proc. Int. Conf. on Foundations of Data Organization (Invited Talk), Kyoto, Japan, May 1985, pp. 21-31

MBW80 J.Mylopoulos, Ph.A.Bernstein, H.K.T.Wong.: A Language Facility for Designing Database-Intensive Applications, ACM Trans. on Database Systems, Vol.5, No.2, June 1980, pp. 185-207

PA86 P.Pistor, F.Andersen: Designing a Generalized NF^2 Data Model with an SQL-type Language Interface, Proc. VLDB, Kyoto, Japan, Aug. 1986, pp. 278-285

PaSc87 H.-B.Paul, H.-J.Schek, M.H.Scholl, G.Weikum, U.Deppisch: Architecture and Implementation of the Darmstadt Kernel System. Proc. SIGMOD '87, San Francisco, pp. 196-207

Pi87 P. Pistor: The Advanced Information Management Prototype: Architecture and Language Interface Overview, 3. Journées de Base Données Avancées, Port Camargue, France, May 1987; also: IBM Heidelberg Scientific Center Tech. Rep. TR87.06.004, June 1987

PT86 P.Pistor, R.Traunmüller: A Database Language for Sets, Lists, and Tables. Information Systems Vol. 11, No. 4, pp. 323-336

RKB85 M.A. Roth, H.F.Korth, D.S.Batory: SQL/NF: A Query Language for ¬1NF Relational Databases. Deptm. Comp. Scienc. Univ. of Texas, Austin, TR-85-19, Sept. 1985

RoSt87 L.A.Rowe, M.Stonebraker: The Postgres Data Model, Proc. VLDB, Brighton, U.K., Sept. 1987, pp. 83-96

Sch86 P.Schwarz, W.Chang, J.C.Freytag, G.Lohman, J.McPherson, C.Mohan, H.Pirahesh: Extensibility in the Starburst Database System, IBM Almaden Research Center, San Jose, Cal., RJ 5211 (54671), 1986, also in Proc. 1986 IEEE Intern. Workshop on Object Oriented Database Systems, Pacific Grove, pp. 85-93

Schm77 J.W.Schmidt: Some High Level Language Constructs for Data of Type Relation, ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977, pp. 247-261

SS86 H.-J.Schek, M.Scholl: The Relational Model with Relation-Valued Attributes, Information Systems 1986, Vol.11, No.2, 1986, pp. 137-147

St84 Stonebraker,M. et al.: Quel as a Data Type, Proc. ACM SIGMOD Conf., Boston, Mass., June 1984, pp. 208-214

St86a M.Stonebraker: Inclusion of New Types in Relational Data Base Systems, Proc. Second Intern. Conference on Data Engineering, Los Angeles, Feb. 1986, pp. 262-269

St86b M.Stonebraker, L.A.Rowe: The Design of Postgres, Proc. ACM SIGMOD Conf., Washington,DC 1986, pp. 340-355

St87 M.Stonebraker, J.Anton, E.Hanson: Extending a Database System with Procedures, ACM Trans. on Database Systems, Vol.12, No.3, Sept. 1987, pp. 350-376

To76 S.J.P. Todd: The Peterlee Relational Test Vehicle - A System Overview, IBM Systems Journal, Vol. 15, No. 4, 1976, pp. 285-308

VKC86 P.Valduriez, S.Khoshafian, G.Copeland: Implementation Techniques of Complex Objects, Proc. 12th Intern. Conf. on Very Large Data Bases, Kyoto, Japan, August 1986, pp. 101-110

Wi83 N.Wirth: Programming in MODULA-2, Springer Verlag 1983