A Data Abstraction Approach to Database Modelling

B. Leavenworth
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598

Introduction

Attempts have been made for some time to reconcile the notions of data base modelling and data abstraction. Considering the overlapping concepts of information hiding and encapsulation from the data abstraction world, and data independence from the database world, it should not be necessary to design yet another programming language as others have done, specialized to a particular data model. Instead, the starting point for our work has been the proposition that an extant general purpose language providing data abstractions should be able to accommodate the popular data models by serving both as a data definition and manipulation language. The criticism has been made that while abstract data types hide the representation details, they also suppress the semantic structure of the data. While this may be true for "programming in the small" [2], it is not the case for "programming in the large" [2]. We will briefly indicate how a CLU-like language [4] (hereafter called XPLS) with minor extensions, plus its supporting module interconnection language (hereafter called the External Structure) can be used as a database definition and manipulation language. XPLS has been designed as a front end to PL/I and is supported by a preprocessor to the PL/I compiler. It turns out that XPLS plus External Structure supports and meshes more smoothly with a semantic data model (for example: [1], [3],[5], [6]) than with the older data models. Our work differs from a number of recent specialized languages which exploit data abstractions and strong type checking but which are based on the relation as a primitive data type. Our approach is not based on any particular data model.

Modelling with Data Abstractions

The External Structure (programming in the large) allows the designer to specify for each data type, the operations characterizing the type, and for each operation, the operation name, types of its parameters and return type (if any). For the purpose of describing data models, the operations are used as (value returning) access functions which are applied to instances of a data type and return either scalar objects or instances of other types. This mechanism handles aggregation [6] quite readily. To model a database, our approach involves defining n+1 data abstractions, where n abstractions correspond to the entity types of the application. The characteristic operations correspond to the attributes or functions in a functional data model (when used in this way, the approach resembles most closely the DAPLEX data language [5]). The definition of these abstractions corresponds to a conceptual schema for a particular data base. For example, the definition for an employee type would have the form:

```
TYPE EMPLOYEE
 DEFINES
 ( * := EMPLOYEE
 NAME(*) -> STRING
 SALARY(*) -> INT
 MANAGER(*) -> EMPLOYEE )
```

where the asterisk is used as a shorthand for the employee type.

The n+1st data abstraction defines for the functional model a parameterized set which can be instantiated with any one of the n entity types. The operations of this type generator include the retrieval and data manipulation functions for the functional model. Some of these encapsulated functions are generic (have type valued parameters) in order to provide adequate type checking. We show an abbreviated definition of this set abstraction below (where EACH and FILTER are iterators):

```
TYPE SET<<T:TYPE>>
 DEFINES
```

```
(* := SET<<T>>
SELECT(*,PROC(T) -> BOOL) -> *
SOME(*,PROC(T) -> BOOL) -> BOOL
SLICE<<T1:TYPE>>(*,PROC(T) -> T1)
     -> SET<<T1>>
INSERT(*,T)
DELETE(*,T)
EACH ITER(*) => T
FILTER ITER(*,PROC(T) -> BOOL) => T )
```

Some earlier work was done in describing the relational and hierarchic models as extended types. It was found that they could be supported at the cost of additional features to XPLS.

Extensions Required

The two principal extensions required by XPLS are subtypes and unnamed internal functions. Subtype declarations are made in the External Structure and support the notion [6] of generalization. The requirement for unnamed internal functions arises from the fact that there are no global variables in XPLS and we wish to avoid idiosyncratic syntax such as the **where** or **such that** clauses that characterize some query languages. In other words, we wish to preserve the general purpose nature of the language and not have two languages,e.g. a separate host language and query language. An unnamed internal function is a function with free variables that can be passed to another function or procedure (AI people and other LISP users will recognize the lambda expression). External procedures cannot be used for this purpose. Unnamed internal functions are used heavily as arguments to iterators and filter (selection) functions. This gives more flexibility to iterators and avoids the rigor mortis of specialized concrete syntax. For example, one iterator can be defined to return just those elements of a collection satisfying a predicate argument (unnamed internal function); another can terminate the stream based on a different predicate argument.

An example which shows a query written in XPLS to print the names of all employees who earn more than their managers is given below. It uses the definition of employee type given previously and makes use of an unnamed internal function.

```
DCL EMPLOYEES SET<<EMPLOYEE>>;

FOR E IN FILTER(EMPLOYEES,
     (EM:EMPLOYEE -> BOOL;
     SALARY(EM)>SALARY(MANAGER(EM))));
   CALL PRINT(NAME(E));
END;
```

The FOR statement works essentially as follows. The unnamed internal function which performs the role of a predicate has a single parameter EM of type EMPLOYEE and returns an object of type BOOL (Boolean). The body of the function appears after the semicolon. The FILTER function, which is an iterator, will select just those EMPLOYEES that satisfy the predicate and bind them one at a time to the variable E. Each time the iterator yields an EMPLOYEE, the body of the FOR statement, which is the print statement, is executed.

Database Interface

The XPLS language has been integrated with a particular database system called NDB [7] which has a data architecture which supports the functional data model, and which provides a PL/I data sublanguage. The interface between the XPLS data manipulation routines and PL/I data manipulation routines in the NDB data sublanguage consists of a set of interface procedures. These interface procedures are written in a dialect or adjunct of XPLS called IPLS. The purpose of an IPLS procedure is to allow an "escape" into PL/I code while preserving the type integrity of the overall program. That is, this approach guarantees that an IPLS procedure cannot violate the data space of XPLS programs.

IPLS procedures are called by XPLS procedures (only abstract scalar arguments may be passed) but not vice versa. Two transfer functions having single parameters are provided: AC (Abstract to Concrete) and CA (Concrete to Abstract). The idea is to transform abstract scalar arguments to PL/I scalar types which can then be manipulated by standard PL/I code. Abstract scalar values are finally passed back to the calling procedure by using the appropriate transfer function. IPLS procedures are type checked in the same manner as for XPLS procedures.

Conclusions

We would conclude by stating some of the advantages to be derived by the kind of merger described above:

The strong typing properties of XPLS are carried over to the realm of database manipulation operations and provide useful consistency checking.

A general purpose data abstraction language becomes by extension an integrated database programming language; the treatment of program and database objects is homogeneous.

The language is not limited to or biased towards any particular data model.

Acknowledgements

I am indebted to Jerry Archibald, Hamed Ellozy, Leigh Power and Bob Taylor for valuable comments and suggestions regarding the work described here.

References

[1] Chen,P.P.S., "The Entity-Relationship Model: Toward a Unified View of Data", *ACM TODS* (March 1976), 9-36.

[2] DeRemer, F. and Kron, H.,"Programming-in-the-Large versus Programming-in-the-Small, *SIGPLAN Notices* (June 1975), 114-121.

[3] Hammer, M. and McLeod, D.,"The Semantic Data Model: A Modelling Mechanism for Database Applications", *ACM SIGMOD* 1978, 26-35.

[4] Liskov, B.H. et al,"Abstraction Mechanisms in CLU",*CACM 20,8* (Aug. 1977), 564-576.

[5] Shipman, D.,"The Functional Data Model and the Data Language DAPLEX", *ACM TODS*, 1980 (to appear).

[6] Smith, J.H. and Smith,D.C.P.,"Database Abstractions: Aggregation and Generalization", *ACM TODS* (June 1977), 105-133.

[7] Winterbottom,N. and Sharman,G.C.H.,"NDB: Nonprogrammer Data Base Facility", Technical Report TR.12.179 (Sept. 1979),IBM United Kingdom Laboratories Ltd.