Types, Algebras and Modeling

Stephen N. Zilles
IBM Research Laboratory
San Jose, CA 95193

Programming languages, database systems and artificial intelligence systems all have the notion that entities can be classified into *types*. As might be expected, however, the usage of the notion of type is not the same throughout or even within these areas. In this paper, I propose a notion of typing that is derived from work on programming languages and indicate how this notion might be used in the context of database systems. Thus, the paper is a step toward unifying the notions of type in databases and programming languages.

Modeling "Things"

The approach to types proposed herein grew out of work on data abstraction in programming languages<LZ74,LZ75>. The purpose of a data abstraction is to model a "thing" without specifying more of its structure and properties than should be externally visible. My view of a "thing" is that it has two aspects: (1) an *identity* which is invariant over time and allows one to investigate the same "thing" at different points in time and (2) a *substance* which may be investigated. I mean substance in a fairly broad sense including both content and structure. Typically the means of investigating the substance of a "thing" is to perform some applicable operation on the "thing" or to send it one or more messages and interpret the responses that are returned. To avoid having to say both operation and message in the sequel, I will only write operation but it should be understood that "operation" includes message oriented protocols as well. In this context, an operation may simply return a property of or piece of the "thing" or the operation may change the nature (or state) of the "thing".

This characterization of "thing", although terribly vague, places the emphasis on the ways in which a "thing" can be investigated rather than on how it should be represented, its behavior rather than its form. With this view, modeling consists of identifying the possible ways of investigating and interacting with the "thing" (operations or messages) and realizing these in some modeling system (a programming language, a first order theory,...). The operations identified in the modeling process are called the operations *applicable* to the "thing" being modeled.

Realizations (models) may take one of two forms: *characterizational* and *representational*. Characterizational models are defined by listing the properties the "thing" has in the model. These properties can be given abstractly, say in the form of axioms obeyed by the operations used to interact with the model of the "thing" or in terms of relationships that must hold between "things". Representational models are defined by choosing some representation for the "thing"s of interest and defining the operations used to interact with "thing"s in terms of their effects on this representation. Typically, a given set of operations are introduced with the representation. If these are the only operations with access to the chosen representation then this representational model is called an *encapsulation*. This paper will focus on representational models.

Types, Algebras and Modules

For this paper, the category of "things" will be restricted to the (sub-)category of data entities. The category of data entities is composed, loosely, of the set of "things" that can be computed and manipulated by a computer. The function of the notion of "type" is to further subdivide this category into distinguishable classes.

The way in which entities are classified into types depends on the purpose of the type classification. In a broad sense, all entities belonging to a type share a common set of properties. In practice, the set of properties that can be used to distinguish types is restricted (a) to fit the purpose for which types were introduced and (b) to allow simple testing for membership in a type.

In programming languages, the most common purpose for classifying entities into types is to be able to verify that, when an operation is applied to an operand, then that operand is among the set of entities (type) that the operation expects. Here, "expects" means that a response is defined for that operand and does not means that the operation necessarily returns a result for that operand. For example, applying the operation TOP to an empty stack of integers may not return an integer but may, instead, cause the signalling of an exceptional condition. The point is that the algorithm that realizes the operation TOP is written to expect an empty stack. It is not, however, written to expect an integer rather than a stack of integers. Thus, types are used to specify the expectations or domain of definition of operations.

Type: One set of objects or values together with a collection of operations, some of which produce elements of the set and (possibly) others which use the elements of the set. These operations may also use or produce elements of other types. Two types are equivalent if the underlying sets and the operations upon them can be put into one-one correspondence such that corresponding operations behave the same way.

As indicated above, operation is used in a very broad sense. It includes what are commonly viewed as operations, such as addition on integers and insertion in data structures, It also encompasses the notion of a property or attribute (of a "thing") which is sometimes <DMN68> considered to be distinct from the notion of operation. Accessing and modifying the properties (query and update in database terminology) are the operations corresponding to the properties.

As is noted in the definition, these operations typically have operands and produce results in a number of distinct types; for example, the operation HAS(element,set_of_elements) yields a boolean result. Therefore, the semantic definition of a type

can often only be given in the context of (a family of) other types.

Algebra: A family of types whose operations are closed within the family. More commomly, an algebra is a family of sorts (the sets underlying the types) together with a collection of operations on the family of sorts (the operations of the types). Each operation is "typed" in that the sorts or types of its operands and results are specified.

In addition to the abstract concepts above, there is the notion of packaging the model (realization) in separate pieces or *modules*. This notion is particularly relevant in structuring the output of programming languages.

Module: An encapsulated implementation of (a) one or more types and/or (b) one or more procedures or functions. In the case of types, the module defines a representation for the elements of each of the types implemented in the module and procedures that implement each of the operations applicable to the types.

Data Abstraction and Data Base Models

In contrast to the viewpoint outlined above, the emphasis in most of the work on data models has been on providing a standard view of all data entities. The common data base models have a standard representation format or structure for entities, the properties of those entities and the relationships among them. The model for a data entity is constructed from a relatively small set of standard modeling primitives. For example, in the relational model, a model is constructed from domains and relations. With each model, there is a standard set of operations, such as create_entity, update_entity, delete_entity, find_entity, which are applicable to all data entities representable in that data model. Those operations that are unique to a particular class of data entities are most often not explicitly identified. This approach is used because it is difficult, if not impossible to predict all the queries which might be asked against the data. Providing a general purpose structure plus a language for combining fragments of the total collection allows for such unanticipated queries.

In many cases, the operations applicable to the entities being modeled translate quite directly into operations on the representation of those entities; for example, changing the value associated with a property of an entity is done with the database update operation. There are, however, cases where neither the representation nor the operations fit naturally into the data model. Consider, for example, the class of queue entities with operations to add, remove and rearrange the queue elements. Representing the queue requires choosing a representation for the elements and the way the elements are ordered. In a relational model, the queue elements might be represented by tuples with a domain for an element identifier and domains for the element contents. The ordering might be represented by an extra ordering domain in the element tuple or by a separate relation consisting of pairs of element identifiers that precede/succeed each other in the ordering. The operations on queues are not simple query language statements, but must be "programs" which take into account the ordering representation.

From the viewpoint of the queue, the form of the queue elements is specified as an external property of the queue, but the form chosen to represent the queue ordering is not to be externally visible and is of concern only to the realization of the queue operations. Thus, the above representations have two distinct portions: the representation of externally defined entities that occur in the entity being modeled and the representation of the details of the model. In choosing a relational representation for queues, these two aspects of the representation get mixed up and both are exposed. The relational representation is not directly a model of the queue; it allows too many extraneous possible interactions. It becomes a model of the queue only when access to it is restricted to the applicable operations. The essence of this point is that representation is not modeling and that the data model operations only sometimes model the operations applicable to the entity being modeled.

The database community recognizes that more than the substance representation is needed to model an entity. This can be seen in recent work on data models <CD79, SM77a, SM77b> where more declarative information is being captured in the model so that the standard operations on the data model, such as delete, will more accurately reflect the semantics of the entity being modeled; for example, when an entity is modeled by several "normalized" components, the normalization information is used to insure that all the components are deleted when the top level component is deleted. This approach increases the number of entities that have natural models in the data model, but it seems unreasonable to believe that, even with such semantic additions, any fixed set of data representations will give natural models for all entities. This is not even claimed by all designers of data models.

Operations versus Constraints

The database community's solution to the overspecification implicit in the data model representation is to augment the chosen representation with a set of integrity constraints. The idea of an integrity constraint is that the standard data model operations will be allowed only when the result will satisfy a set of constaints (or invariants) on the form and content of the representations of the entities being modeled. A typical constraint might be that the sum of the salaries of all employees in a department must be less than the total salary budget for that department. Then an update operation to increase an employee's salary would be allowed only if the departmental budget constraint was met.

The notion of constraint is very much the same as the notion of invariants in the implementation of data abstractions in programming languages. Both constraints and invariants define a subspace of the modeling space which is sufficient to represent the entities being modeled. They differ in that the satisfaction of the invariant is established with respect to packaged sets of data model operations that realize the operations that are applicable to the entities; that is, the satisfaction of the invariants is established once and for all and need not be checked again. In contrast to this, the constraints must be checked many times, after each set of related operations on the representation. This is not to say that the constraints (nor the invariants) must be satisfied at every point in time. They need only be satisfied between (at the begining and end of) related sets of database operations. A related set of operations is called a *transaction* and it is the analogue of the data abstraction operation. Given this similarity and the greater efficiency of verifying the satisfiability of invariants, it makes sense to encapsulate the representation in the data model with a set of operations that define a data abstraction whenever this is possible.

To preserve the capability of handling unanticipated queries, this encapsulation need not be total. To insure the constraints are satisfied, it is only necessary that all the operations that change the state of the representation be put in the encapsulation. It is possible and appropriate to let some or all of the standard data model operations for querying show through the encapsulation in a semi-transparent way. Semi-transparency means that some of the operations applicable to the representation of a data abstrac-

tion are exported directly as operations applicable to the data abstraction itself.

Adopting any one of the data models proposed to date is equivalent to picking a particular fixed set of abstractions which will then be used to model all entities. As the queue example indicates, it is likely that, given any fixed set of (abstract) representational constructs, there will always be entities that have no natural model in that set. Therefore, one should also have a facility for defining encapsulated data abstractions to handle these exceptional situations.

Putting Data Abstractions in a Data Base

Allowing users to extend the collection of types via a data abstraction encapsulation mechanism forces extentions to many of the existing data modeling approaches, at least as they are realized on computer systems. Instead of having a fixed set of data types for the values of properties (i.e. fields in records or tuples), the data model must allow an open ended set of domain types. This forces modifications to both the storage of basic units such as records, tuples and segments, as well as forcing extensibility in the area of indices on these new kinds of domains. An interface is probably required to add new types to the collection of recognized domain types.

Data Models as Data Abstractions

Data models are "thing"s in their own right and, as such, are appropriate to model as particular data abstractions. Doing so introduces a number of interesting problems that do not occur in the examples commonly found in programming language treatments of data abstraction. Perhaps the most important problem is that it is quite typical to have multiple representations of a single data abstraction present in the data base system. This is a direct consequence of the principle of *data independence* which specifies that the user's view of the data should not have to change because the underlying representation has been changed to improve the efficiency of access. This implies that there must be a fairly sophisicated package of representation translation tools within the data model abstraction.

Subtypes, Unions and Generalizations

In modeling the world it is often common to find that a entity, such as a person Harry Jones, belongs to more than one class (type) and has properties (operations) that are specific to its (his) membership in each class. For example, Harry Jones may be a Student at a university and also be a part-time Employee of the university. In both of those roles he is a Person known to the university. This collection of classes to which an entity belongs is often called the *generalization* structure. Contained classes or types, such as Student and Employee, are called *subtypes* of the containing class (Person). A generalization (or supertype) is defined by choosing a subset of the operations applicable to the class (type) of entity being generalized. (Recall that operations as used here subsume properties of entities.) The generalization Person may have operations to obtain a residence address, a telephone number, etc.. These are a subset of the operations applicable to Students which would also have operations to obtain and modify grades and other student properties. Clearly, multiple generalizations on the same class are possible by choosing to preserve different subsets of the operations.

There is some similarity between the notion of generalization as described above and the notion of *union* types in programming languages. Both have a notion of subtype or, perhaps more accurately, types embedded in other types. The difference is that generalizations as defined above are open ended; if a new type that has the operations required by the generalization is defined, then it automatically becomes a subclass of the generalization class. Unions, on the other hand, are composed from a finite collection of classes that are explicitly identified. It is also the case that the classes in a union may have no operations in common. Unions are most useful in solving representation problems; for example, in defining the class of lists it is useful to have different representations for empty and nonempty list. The list representation would then be the union of the two subcase representations.

Summary

Data abstraction as it is practiced in programming languages is a notion distinct from the notion of data model in databases and is complementary to it. It makes sense to augment a data model (network, relational, hierarchic, entity-set) with a facility for defining encapsulated data abstractions to handle modeling entities that do not have natural models in the data model. Encapsulation is also useful in reducing the cost of maintaining constraints on a data model. In some cases, encapsulation can completely eliminate execution time checking of constraints. There are, however, several problems, such as multiple representations, storing data abstractions in the data model, and generalization that still need satisfactory solutions if data abstraction is to be effectively used in the database enviroment.

Acknowledgments

The views presented here were in part formed during discussions with Rudolf Bayer and Herbert Weber. Drafts of the paper were read and improved by Peter Lucas and Bryant York. I, of course, take full responsibility for any errors or misconceptions.

References

<CD79> Codd, E. F., Extending the Database Relational Model to Capture More Meaning, IBM Research Report, RJ2599, August 1979.

<DMN68> Dahl, O-J., B. Myrhaug and K. Nygaard, *Simula 67 Common Base Lanuage*, Norwegian Computing Center, Oslo (1968).

<LZ74> Liskov,B., and Zilles,S., "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4, (April 1974), 50-59.

<LZ75> Liskov,B., and Zilles,S., "Specification Techniques for Data Abstractions, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, (March 1975), 7-19.

<SM77a> Smith, J. M. and Smith, D. C. P., "Database Abstractions: Aggregation," *CACM*, Vol. 20, No. 6, (June 1977).

<SM77b> Smith, J. M. and Smith, D. C. P., "Database Abstractions: Aggregation and Generalization," *ACM TODS*, Vol. 2, No. 2, (June 1977).

<TWW78> Thatcher, J. W., Wagner, E. G. and Wright, J. B., "Data Type Specifications: parameterization and power of specification techniques," *Proceedings*, SIGACT 10th Annual Symposium on Theory of Computing, (May 1978), 119-132.