Multiple Inheritance and the Closure of Set Operators in Class Hierarchies

John L. Pfaltz
James C. French

IPC-TR-92-004
June 25, 1992

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 2290l

This research was supported in part by
DOE Grant #DE-FG05-88ER25063 and
JPL Contract #957721

Abstract

In this report, we establish essential closures in class hierarchies of database systems that support set operations, such as union and intersection, in their query language. In particular, we rigorously demonstrate that multiple inheritance is an implementation requirement, as is the formal treatment of the class hierarchies as lattices with defined least upper, and greatest lower, bound operators.

Multiple Inheritance and the Closure of Set Operators in Class Hierarchies †

John L. Pfaltz
James C. French

Institute for Parallel Computation
University of Virginia, Charlottesville VA

Two capabilities seem essential in new database implementations — class inheritance and set operations. In the course of designing and prototyping ADAMS, a distributed database language developed in the Institute for Parallel Computation at the University of Virginia, we sought implementation semantics for both of these key concepts. They are non-trivial; our initial intuitive semantics turned out to be flawed.

We began by assuming an underlying, object based [Weg87] *entity database model* which is compatible with extended entity-relationship models [Che76, TYF86], many semantic models [AbH87, PeM88], and to which the relational model can be easily extended by adjoining a unique symbolic identifier to every tuple — as is common in many implementations. In short, this entity model, developed in the remainder of this section is intended to be a purely vanilla model with no surprises, of which more practical systems are refinements. On this we define (in Section 2) the concept of the "compass" of a class, which then forms the basis of our set operation semantics (in Section 3). One consequence of this semantic analysis will be the need for multiple inheritance [AbH87, Car84, Tou86], even though, as Peckham and Maryanski [PeM88] note, it "can be difficult to control". Its control can be facilitated [PFG91] by implementing classes as entities themselves in well-defined lattices with the associated least upper bound, and greatest lower bound, operators.

1. Entity Database Model

The key presupposition of the entity database model is that the database is implemented by creating uniquely identifiable entities. We impose a type structure on entities, or objects, by assigning them to a *class.* All entities in the class, for example the class PERSON, will share common properties, such as *name, home_address, age,* and *social_security_number.* We will also assume the existence of subclasses. Most semantic and object-oriented databases use an *IS_A* construct to support the concept of class and subclass.

Brachman [Bra83] correctly notes that *inheritance* as defined by the *IS_A* construct is really little more than convenient syntactic shorthand for incrementally creating subclasses, so we will ignore actual inheritance mechanisms *per se.* The important feature that is abstracted in the entity database model is the very existence of classes and subclasses, which can be declared by what ever syntactic formalism is convenient. For example, the class DOCTOR, with the additional properties *speciality, training,* and *office_address,* might be a subclass of PERSON. The class PERSON may also subsume a subclass PATIENT, with the additional properties *case_history, complaint,* and *outstanding_amount_due.*

In these preceding examples, classes and subclasses have been characterized by properties which are called *attributes* in both relational and object-oriented terminology. For our purposes, we will assume that a class attribute is a *singled valued* function, f, of a single variable x where x denotes the unique identifier of an entity instance within the class. The image of f may be a value (e.g. printable string or icon), or some other entity. Permitting f to be set-valued as in DAPLEX [Shi81] will not alter the generality of our approach.

In most database systems, classes are defined in terms of their associated attribute properties. But they can also be defined by imposing predicate restrictions on class membership. For example, we might choose to declare JUVENILE to be a subclass of PERSON, with the restriction that $age < 21$.

The preceding intuitive introduction to classes, subclasses, and a class hierarchy in the entity database model can be made more formal. Let $F = \{ f_i \}$ denote a set of functions associated with a particular class C. Following the syntax used in ADAMS [PSF88] we will use the expression $x.f_i$ to denote the image of x under f_i. For all $x \in C$ the attribute expression $x.f_i$ is said to be *meaningful* for any $f_i \in F$, even though its actual value, or image, may as yet be undefined. That is, all attribute functions are assumed to map into a domain to which an additional *undefined* value has been adjoined.

By a class restriction, we shall mean an expression E in the predicate calculus with exactly one free variable. The expression $V.age < 21$ is an example. (Here we capitalize the free variable for visual emphasis.) The expression is evaluated by replacing the free variable with an entity instance x to determine if it can belong to the class.

A class is defined by its set F of associated attribute functions and a restricting expression E. That is, $C = (F, E)$. F may be empty and E may be omitted. In the latter case $x.E = true$ vacuously. It will be convenient to assume a meta-linguistic operator *class_of* which given a specific instance x of an entity belonging to a class C denotes its class. That is, *class_of(x) = C,* $\forall\, x \in C$.

A class $C_i = (F_i, E_i)$ is said to be a *subclass* of $C_k = (F_k, E_k)$, denoted $C_i < C_k$, if

(a) $F_i \supseteq F_k$, and

(b) $E_i \rightarrow E_k$ (that is, E_i logically implies E_k).

Condition (a) seems to be universally accepted in both object-oriented and semantic-network class hierarchies; and we would assert that a condition similar to (b) is also essential since membership in the subclass C_i must imply membership in its superclass C_k. Given this definition, it is easy to show that

Proposition 1.1: < is a partial order on any collection of classes.

We will now say that a database implementation belongs to the *entity database model* if the implementation supports

(a) uniquely identifiable entity instances;

(b) a class hierarchy including on attribute properties and/or predicate restrictions; and

(c) the standard set operations (e.g. union, intersection, relative complement) on sets of entities.

The unique identifiability of any entity instance is an important formal characteristic of the entity database model. However implemented—it could be a literal storage address, a symbolic string, or a functional accessing mechanism—the unique identifier which we denote by lower case letters *x, y* or *z,* is not an attribute of the instance it identifies. In the entity database model, two distinct entity instances x and y belonging to a single set of instances of class C may be functionally identical in all respects, that is we may have $x.f_i = y.f_i$ for all $f_i \in F_C$. This is impossible in the relational model, in which any two tuples belonging to a single relation r with schema F_R must at least differ over some set of key attributes $K \subseteq F_R$. Khoshafian and Copeland give a general discussion of object identity in [KhC86]. Some of the issues encountered in implementing an entity naming paradigm are discussed in [PFW88]. In this report, the precise mechanism used to identify entities is not at issue.

In the entity database model, one deals with sets of entities. These sets, which we will denote by the uppercase letters *X, Y,* and *Z,* are themselves entities and so must belong to some class which is distinct from the class of their constituent entities. We require a class constructor of type SET which defines a new class, denoted by $S[C]$, of sets whose elements belong to the class C. Many type theories introduce class constructors based on Cartesian product (for tuples) [HuK87] or disjoint sum (for variant records) [AtB87]. It is our belief that class extension using a set constructor is conceptually simpler.

It may be the case that $F_{S[C]} = \emptyset$; although a subclass of $S[C]$ might have attribute functions associated with the set as a whole, distinct from the individual elements. But $E_{S[C]}$ can not be omitted. $E_{S[C]}$ must include the predicate

$$(\forall x \in V)\ [\ \text{class\_of}(x) = C\ ]$$

as one of its conjuncts. Here V is the free variable. The expression is evaluated by replacing V with a specific set instance, so that if X is an instance set in $S[C]$ then $(\forall x \in X)$ [class_of$(x) = C$].

Proposition 1.2: $S[C_i] < S[C_k]$ **implies that** $C_i < C_k$**. Further, if** $S[C_k]$ **has no additionally declared attributes or predicate conjuncts, then** $C_i < C_k$ **implies** $S[C_i] < S[C_k]$**.**

Proof: If $S[C_i] < S[C_k]$ then $E_{S[C_i]} \rightarrow E_{S[C_k]}$, hence $(\forall v \in V)$[class_of$(v) = C_i$] $\rightarrow$ $(\forall v \in V)$[class_of$(v) = C_k$]. Since V can range over all sets of C_i elements, $x \in C_i$, implies $x \in C_k$.

Conversely, if $C_i < C_k$ then $(\forall v \in V)[\text{class\_of}(v) = C_i] \rightarrow (\forall v \in V)[\text{class\_of}(v) = C_k]$. If $E_{S[C_k]}$ has no additional conjuncts, then $E_{S[C_i]} \rightarrow E_{S[C_k]}$; and if $F_{S[C_k]} = \emptyset$, the conditions for a subclass are satisfied. □

This proposition emphasizes the apparent isomorphic correspondence between the class hierarchy of sets of elements and the class hierarchy of their constituent elements, as is illustrated in Figure 1-1, provided no additional attributes or restrictions are associated with these set classes.

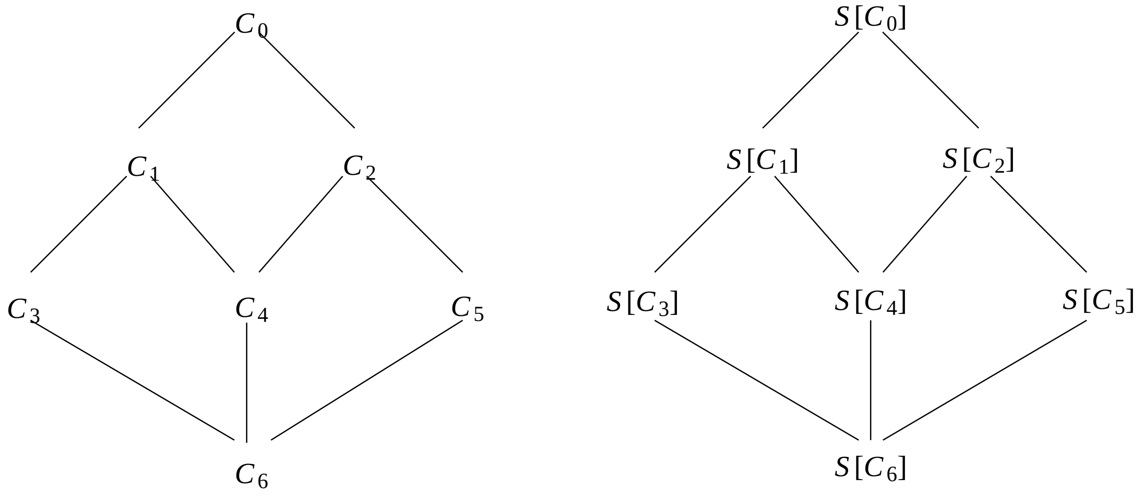

Figure 1-1.

However, none of the following results which assert necessary properties of the element classes, C_i, depend on this isomorphism property.

2. The Compass of a Set

A class, $C = (F, E)$, can be regarded as either a generic template by which class membership is defined, or as the conceptual set of all possible entities which could belong to the class, that is all conceivable x over which all of the $f_i \in F$ are defined and $x.E$ is true, whether instantiated or not.[1] It will be convenient to refine the idea of a conceptual set of possible entities by introducing a formalism called *compass*.

[1] In some object-oriented implementations [GoR83, Kim89], the class (through its class manager) also denotes all actually instantiated elements as well.

By the *compass* of an expression, denoted **comp**(exp), we mean the conceptual set of all entities which could possibly satisfy the expression; that is, the abstract set of all entities which could be *encompassed* by exp. For example,

$$\mathbf{comp}(F) = \{ x \mid \text{the expression } x.f_i \text{ is meaningful for all } f_i \in F \}, \text{ and}$$

$$\mathbf{comp}(E) = \{ x \mid x.E = true \}.$$

The compass of such expressions is typically an infinite set, and hence tends to be of theoretical rather than operational interest.

Proposition 2.1:
If $F_i \supseteq F_k$, then comp(F_i) $\subseteq$ comp(F_k).
If $E_i \rightarrow E_k$, then comp(E_i) $\subseteq$ comp(E_k).

Proof: Let $x \in$ **comp**(F_i) implying that $x.f$ is valid for all $f \in F_i$. Since $F_k \subseteq F_i$, if $f \in F_k$ then $x.f$ must be valid, so $x \in$ **comp**(F_k).

Since if $x.E_i$ is true, $x.E_k$ must be true by implication, the second containment follows immediately □

The inversion of containments of the first assertion has also been noted in [ACO85].

The compass of a class expression, **comp**(C), is then just the class itself, since any x in **comp**(C) must have all $f_i \in F_C$ associated with it, and must also satisfy any restricting expression E. Further, it is evident that

$$\mathbf{comp}(C) = \{ x \mid \text{class\_of}(x) = C \} = \mathbf{comp}(F_C) \cap \mathbf{comp}(E_C).$$

Recall that C_i was said to be a subclass of C_k if $F_i \supseteq F_k$ and $E_i \rightarrow E_k$, so we immediately have

Proposition 2.2: If C_i is a subclass of C_k then comp(C_i) $\subseteq$ comp(C_k).

This accords completely with our intuition. The set of entities which could belong to the class DOCTOR must be contained in the set which could belong to the class PERSON.

3. Set Operators in a Class Hierarchy

Given specific instance sets, the fundamental set operators union ($\cup$) and intersection ($\cap$) must be supported. One can adopt implementation semantics which require that the elements of the operand sets be of precisely the same type, or class, in order for the operator to be defined. For example, in a strongly typed language, such as Pascal, one can not apply the boolean AND operator to two operands, one of which is boolean and one of which is real, because it makes no sense. However, it is also customary to loosen the implementation semantics whenever possible, provided that the type of the result is meaningful. Thus in Pascal, it is legal to perform arithmetic operations (+, -, *, /) even though the operands are of mixed real and integer type—the result is assumed to be real. (Notice that one could regard the class *integer* as a subclass of the class *real*, and that **comp**(integer) $\subset$ **comp**(real).)

The concept of a union of a set of doctors and a set of patients, regarded simply as people clearly makes sense. Cardelli and Wegner call this *inclusion polymorphism* in [CaW85]. Entities of type doctor can always be regarded as being of the form PERSON. Similarly, the concept of the intersection of a set of doctors and patients to denote those persons who are both DOCTORS and PATIENTS also makes polymorphic sense. We seek implementation semantics that will be consistent with the class hierarchy.

3.1. Set Union

Suppose that the operation X $\cup$ Y is well defined. To what class should the elements of the resultant, $X \cup Y$, belong?

Let X be an instance set in $S[C_1]$ and let Y be a set in $S[C_2]$. X $\cup$ Y will be an element of $S[C_{1\cup 2}]$, where $C_{1\cup 2}$, denoting the class of elements in X $\cup$ Y, is to be defined. Readily, if $C_1 = C_2$ then $C_{1\cup 2} = C_1 = C_2$, and if $C_1 < C_2$ then $C_{1\cup 2} = C_2$. For the most general case assume that C_1 and C_2 are non-comparable classes.

There are at least three different possible implementation semantics to be considered.

Option 1: $F_{1\cup 2} = F_1 \cup F_2$, $E_{1\cup 2}$ = { any definition }.

Since $F_{1\cup 2} = F_1 \cup F_2 \supseteq F_1$, **comp**($X \cup Y$) = **comp**($F_1 \cup F_2$) $\subseteq$ **comp**(F_1) = **comp**(X). However, this contradicts our understanding of the union operator for which one expects **comp**(X) $\subseteq$ **comp**($X \cup Y$).

Option 2: $F_{1\cup 2} = F_1 \cap F_2$, $E_{1\cup 2} = (E_1 \vee E_2)$.

The problem with this semantic interpretation of the union class is that $E_{1\cup 2} = E_1 \vee E_2$ need not be well defined on $F_1 \cap F_2$. That is, $F_{E_1 \vee E_2}$ may not be contained in $F_1 \cap F_2$. Note that only functions applied to the free variable can cause trouble. This leads to the following variation.

Option 3: $F_{1\cup 2} = F_1 \cap F_2$, $E_{1\cup 2} = E'$.

This refinement of option 2 assumes that it is possible to find a maximally restrictive expression E' that is defined over $F_1 \cap F_2$ with the property that $E_1 \rightarrow (E_1 \vee E_2) \rightarrow E'$ and $E_2 \rightarrow (E_1 \vee E_2) \rightarrow E'$. One way of deriving E' is to express $E_1 \vee E_2$ in conjunctive normal form. Those conjuncts which are not well defined on $F_1 \cap F_2$ are discarded to obtain E'.

The implementational semantics associated with the latter option are logically consistent. Since $F_1 \cap F_2 \subseteq F_1$ and $E_1 \rightarrow E'$, we have **comp**$(X) \subseteq$ **comp**$(X \cup Y)$, and similarly **comp**$(Y) \subseteq$ **comp**$(X \cup Y)$. For these reasons we assert that

Proposition 3.3: If $X \in S[C_1]$ and $Y \in S[C_2]$, then $X \cup Y \in S[C_{1\cup 2}]$, where $F_{1\cup 2} = F_1 \cap F_2$, and $E_{1\cup 2} = E'$, where $E_1 \vee E_2 \rightarrow E'$, where E' is the maximally restrictive expression defined over $F_1 \cap F_2$ such that the implication holds.

This is illustrated in Figure 3-1. Note that in the general case, $F_{1\cup 2} = F_1 \cap F_2$ may be empty, in which case E' must be vacuously *true*. Any entity (that is not a set) must belong to such a universal class of all entities (with no declared properties).

In a semantic class hierarchy that has been created by both *specialization* and *generalization*, the proposition above denotes the most general, logically consistent definition of a union class. However,

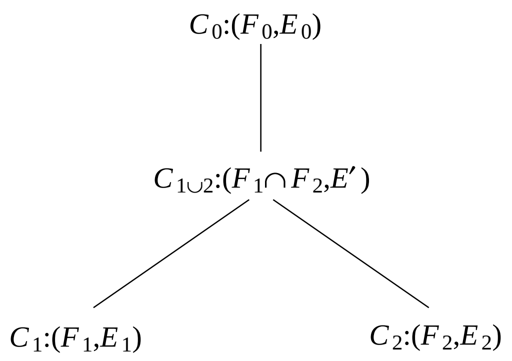

Figure 3-1

there are practical reasons for not supporting generalization as a mechanism for creating new classes, c.f. [Pfa88]. All object-oriented, and many semantic, database systems, including ADAMS, only allow the creation of new classes through specialization, or subclass declaration. In this context there is yet a fourth option for specifying the implementation semantics of the union operator that is also logically consistent, although not completely general.

In this case, we assume that C_1 and C_2 are non-comparable subclasses of C_0 ($C_1 < C_0$ and $C_2 < C_0$) which is the most restrictive super class of both; that is C_0 is their least upper bound in the partial order <.

Option 4: $F_{1 \cup 2} = F_0$, $E_{1 \cup 2} = E_0$.

Since $F_1 \cap F_2 \subseteq F_0$ and $E_1 \rightarrow E_1 \vee E_2 \rightarrow E_0$, we have **comp**$(X) \subseteq$ **comp**$(X \cup Y)$, and similarly **comp**$(Y) \subseteq$ **comp**$(X \cup Y)$, demonstrating that these implementation semantics are, at least, logically consistent. These are the union class implementation semantics that have been adopted by ADAMS.

3.2. Set Intersection

Following the reasoning we employed above regarding the union operator, there is only one plausible choice for the class of resultant elements in the case of the intersection operator.

Option: $F_{1 \cap 2} = F_1 \cup F_2$**,** $E_{1 \cap 2} = (E_1 \wedge E_2)$**.**

Applying the containment relations to the compass concept we obtain **comp**$(X \cap Y) \subseteq$ **comp**(X) and **comp**$(X \cap Y) \subseteq$ **comp**(Y) as we intuitively expect. Moreover, since any element $x \in X \cap Y$ must denote an entity that is in both the instance sets X and Y, it must necessarily have all $f_i \in F_1$ and all $f_k \in F_2$ defined on it. And it must satisfy both restrictions E_1 and E_2.

Thus, we obtain the following proposition and class structure shown in Figure 3-2.

Proposition 3.4: If X belongs to $S[C_1]$ and Y belongs to $S[C_2]$ then $X \cap Y$ belongs to $S[C_{1 \cap 2}]$, where $F_{1 \cap 2} = F_1 \cup F_2$ and $E_{1 \cap 2} = E_1 \wedge E_2$.

Note that (1) given two classes C_1 and C_2, their intersection class is always well defined; and that (2) the schema of a natural join in the relational model (which can be interpreted as an intersection operator) is also $F_1 \cup F_2$.

$$\begin{tikzcd}
 & C_0{:}(F_0,E_0) \arrow[dl, dash] \arrow[dr, dash] & \\
C_1{:}(F_1,E_1) \arrow[dr, dash] & & C_2{:}(F_2,E_2) \arrow[dl, dash] \\
 & C_{1\cap 2}{:}(F_1 \cup F_2, E_1 \wedge E_2) &
\end{tikzcd}$$

Figure 3-2

Intersection classes can be declared at the same time that any two classes C_1 and C_2 are defined, thereby allowing the creation of entity instances belonging to the intersection. But the process is an open ended one. Suppose that we create a third subclass C_3 of C_0. Then to support all possible intersections we would need the class structure as shown in Figure 3-3.

$$\begin{tikzcd}
 & C_0{:}(F_0,E_0) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
C_1{:}(F_1,E_1) \arrow[d, dash] \arrow[dr, dash] & C_2{:}(F_2,E_2) \arrow[dl, dash] \arrow[dr, dash] & C_3{:}(F_3,E_3) \arrow[dl, dash] \arrow[d, dash] \\
C_{1\cap 2}{:}(F_1 \cup F_2, E_1 \wedge E_2) \arrow[dr, dash] & C_{1\cap 3}{:}(F_1 \cup F_3, E_1 \wedge E_3) \arrow[d, dash] & C_{2\cap 3}{:}(F_2 \cup F_3, E_2 \wedge E_3) \arrow[dl, dash] \\
 & C_{1\cap 2\cap 3}{:}(F_1 \cup F_2 \cup F_3, E_1 \wedge E_2 \wedge E_3) &
\end{tikzcd}$$

Figure 3-3

There is no logical reason to resist the construction of such a semi-lattice of class declarations; but to manually declare it is surely tedious. An implementation of an entity database model can do this automatically whenever two or more subclasses are declared. The necessary information is available; and the automatic construction of $C_{i \cap k} : (F_i \cup F_k, E_i \wedge E_k)$ is quite straightforward. To prevent a proliferation of unnecessary class definitions, in ADAMS, we defer actual creation of intersection classes until at least one entity will become a member of the class. Note that the class hierarchy automatically becomes a lattice of data types as described in [Sco76], or more particularly a semi-lattice structure as described in [Ada85], if one does not assume that the least upper bound of two classes need exist, and a distributive lattice if one assumes a universal super class with no attributes or predicates, together with the implicit definition of all intersection classes.

4. Summary

If we seek to implement an entity database model which allows the creation of a class hierarchy through successive refinement, or specialization, as most object-oriented and semantic database models do, and if we expect the implementation to support standard set operations, then certain requirements follow naturally.

1) In addition to base entity classes C_1, C_2, ... C_k there must be corresponding set classes $S[C_1]$, $S[C_2]$, ... $S[C_k]$ characterizing those manipulable entities which are sets of entities.

If, in the implementation language, one strictly restricts set operations to operands belonging to the same set class, then no more is required. But such a restriction is overly stringent. Assuming a more general polymorphic treatment of intersection and union operators as discussed above, we have the following conclusions.

2) In order to obtain closure for the intersection operator, given two non-comparable classes C_i and C_k, one must be able to declare their intersection class $C_{i \cap k}$ and, of course, the associated set class $S[C_{i \cap k}]$ to which the resultant set will belong. Similarly, to ensure closure for the union operator, a

reasonable implementation approach is to let the union class consist of elements whose class $C_{i \cup k}$ is the least upper bound of C_i and C_k. This is not the most general possible implementation.

3) The implementation system must maintain, possibly only implicitly, a semi-lattice of class declarations C_1, C_2, ... C_n, whose greatest lower bound is the intersection class $C_{1 \cap 2 \cap \cdots \cap n}$: $(F_1 \cup F_2 \cup \cdots \cup F_n, E_1 \wedge E_2 \wedge \cdots \wedge E_n)$. A similar semi-lattice of set-class declarations, and possibly of set-set-class declarations must also be maintained.

4) In view of 3) above, a form of multiple inheritance induced by the formation of intersection classes is required. But it need not be of the generality described in [Car84] or [Tou86].

We believe that these represent minimal characteristics which must be present in any entity, or object based, database implementation whose class hierarchy is defined by specialization, or subclass declaration. These characteristics can be implemented [PFG91]. More general implementations, such as one which would support generalization, or equivalently the complete closure of the union operator, seem possible, but computationally complex.

5. References

[AbH87] S. Abiteboul and R. Hull, IFO: A Formal Semantic Database Model, *Trans. Database Systems 12*,4 (Dec. 1987), 525-565.

[Ada85] T. Adachi, Powerposets, *Inf. and Control 66*(1985), 138-162.

[ACO85] A. Albano, L. Cardelli and R. Orsini, Galileo: A Strongly Typed Interactive Conceptual Lanugage, *Trans. Database Systems 10*,2 (June 1985), 230-260.

[AtB87] M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *Computing Surveys 19*,2 (June 1987), 105-190.

[Bra83] R. J. Brachman, What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks, *COMPUTER 16*,10 (Oct. 1983), 30-36.

[Car84] L. Cardelli, A Semantics of Multiple Inheritance, in *Semantics of Data Types, Lecture Notes in CS 173*, Springer Verlag , June 1984, 51-67.

[CaW85] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys 17*,4 (1985), 471-522.

[Che76] P. P. Chen, The Entity-Relationship Model---Toward a Unified View of Data, *Trans. Database Systems 1*,1 (Mar. 1976), 9-36.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

[HuK87] R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *Computing Surveys 19*,3 (Sep. 1987), 201-260.

[KhC86] S. N. Khoshafian and G. P. Copeland, Object Identity, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 406-416.

[Kim89] W. Kim, A Model of Queries for Object-Oriented Databases, *Proc. 15th Conf. on VLDB* , Amsterdam, Aug. 1989, 423-432.

[PeM88] J. Peckham and F. Maryanski, Semantic Data Models, *Computing Surveys 20*,3 (Sep. 1988), 153-190.

[PFW88] J. L. Pfaltz, J. C. French and J. L. Whitlatch, Scoping Persistent Name Spaces in ADAMS, IPC TR-88-003, Institute for Parallel Computation, Univ. of Virginia, June 1988.

[PSF88] J. L. Pfaltz, S. H. Son and J. C. French, The ADAMS Interface Language, *Proc. 3th Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988, 1382-1389.

[Pfa88] J. L. Pfaltz, Implementing Set Operators Over a Semantic Hierarchy, IPC TR-88-004, Institute for Parallel Computation, Univ. of Virginia, Aug. 1988.

[PFG91] J. L. Pfaltz, J. C. French and A. Grimshaw, An Introduction to the ADAMS Interface Language: Part I, IPC TR-91-06, Institute for Parallel Computation, Univ. of Virginia, Apr. 1991.

[Sco76] D. S. Scott, Data Types as Lattices, *Siam J. on Computing 5*,3 (Sep. 1976), 522-587.

[Shi81] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *Trans. Database Systems 6*,1 (Mar. 1981), 140-173.

[TYF86] T. J. Teorey, D. Yang and J. P. Fry, A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, *Computing Surveys 18*,2 (June 1986), 197-222.

[Tou86] D. S. Touretzky, *The Mathematics of Inheritance Systems*, Morgan Kaufmann Publ., Los Altos, CA, 1986.

[Weg87] P. Wegner, Dimensions of Object-Based Language Design, *Proc. OOPSLA '87*, Oct. 1987, 168-182.