Can Object-Oriented Databases be Statically Typed?*

Val Breazu-Tannen Peter Buneman Atsushi Ohori

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389

1 Foreword

Can there be a database programming language with a type system that matches the data model? Can such a language be strongly typed? Can it be statically typed? A static type system for database programming languages is, of course, the Holy Grail of the subject. However we, the authors, who have spent some effort in searching for this venerated object, now doubt its existence and have come to believe that, where object-oriented and semantic data models are concerned, we must either give up the Quest or substantially modify our view of a type system. Worse still, we are not in agreement about which course of action to take.

In the spirit of *glasnost* we decided simply to expose the disagreement. This paper is therefore constructed as an argument how we might resolve the dilemma. We shall start therefore on what we all believe to be our goals and then open the debate. Rather than reveal the identities of the protagonists (whose positions have not always remained consistent) we shall present a set of arguments and counter-arguments. The reader is invited to join in the fray.

2 Introduction

Types arise naturally in our mental processes. Whenever we try to accommodate more information than we can handle at one time, we try to find an appropriate system of classification in order to impose some general structure on the information at hand. Not surprisingly we are naturally led to exploit types both in programming languages and in databases where some form of classification is essential in order to deal with the complexities of the data structures involved. And, since we use programming languages to communicate

* This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634. The third author was also supported in part by OKI Electric Industry Co., Japan.

Published in **Proc. 2nd International Workshop on Database Programming Languages, pages 226–237, Morgan-Kaufmann Publishers, 1989.**

with databases, it is not surprising that we would like the classification systems to be related or, better, the same.

The type systems of programming languages have been fettered with the constraint that they should be implementable and relate to the computation that should be performed. The successful type systems have therefore not only been useful as conceptual structures, but also as an important component of the implementation of the language. On the other hand, type systems for databases – which we usually call data models – have not always been subject to the same constraint. There are a number of data models (see [Hul87]) that have never been fully implemented and yet have been of real value in "conceptual design" – an essential part of database implementation.

The type structure of a programming language goes hand in hand with its collection of primitive operations. The effect of operations is well-defined only on operands of certain types. In fact, it is the operations which define the types, so the types are there. The question is how much recognition we should give them.

One may call a programming language *strongly typed* if, before or during the evaluation of a program, checks are made to ensure that primitive operations always receive arguments of the correct intended type. This is a rather vague definition, since it depends on what we mean by "intended type". Moreover, it begs the question of whether there is a difference between "weakly typed" and "untyped"! Therefore, it seems difficult to settle arguments on whether languages are strongly typed or not. For example, Lisp is sometimes claimed to be a strongly typed language but, after being asked to predict an interpreter's evaluation of the following two expressions,

```
(define TYPE_ERROR (+ (or (< 2 3) 4) 5))
(define  NINE  (+ (or 4 (< 2 3)) 5))
```

one may reasonably question the claim. It depends on whether one believes there is a boolean type.

In any case Lisp and many other languages are *dynamically* checked: the types of arguments are checked as the operators are applied. In *static* type checking, the type consistency can be completely determined before execution. (A more detailed discussion of these concepts can be found in [CW85].)

Static typing has several advantages. It makes programs more reliable catching numerous conceptual errors at compile-time through the type errors that ensue. The type structure of a program reflects, hopefully, some of the structure of the problem solved by the program, so types are a form of obligatory documentation. Assuming we insist on strong typing, as we should, static typing will make programs more efficient by eliminating the need for run-time type checks (while making compiling less efficient, of course). And, finally, on a different level, designing languages with static typing seems more likely to produce clean semantics. Somehow, the temptation for ad-hoc, implementation-driven decisions such as the one exemplified by the `TYPE_ERROR` and `NINE` expressions is less frequent.

Until recently, statically typed languages were justly criticized for being too cumbersome. First, they did not allow certain types of generic code; and second the type declarations, while admittedly useful as documentation, were often tedious and obvious. Both factors caused programs to become bulky and difficult to modify. With the introduction of polymorphism and type inference into languages such as ML [HMT88] and Miranda [Tur85], we believe [BTBG88] that the flexibility of languages like Lisp is now possible in statically

typed languages. All this is not to say that static typing (even more so, type inference) are always possible. Sometimes it may be too restrictive to give types to all the variables at compile-time. In particular, dynamic type checking will be sometimes unavoidable when dealing with persistent data. However, we do not see this as a reason for abolishing static type-checking; instead we should adopt a policy [AB87] of statically type-checking the largest possible "chunks" of programs, in a fashion that would be consistent with the smaller amount of dynamic checking.

In view of the arguments we gave, it is natural to try to integrate static type-checking with the various programming paradigms in current use, most importantly with the powerful ideas underlying object-oriented programming [GB80]. There has been considerable recent research in this direction [Car84b, Mit84, CW85, Wan87, FM88, Car88b, Sta88, Car88a, OB88, JM88, Car88c, Rem89]. It turns out that some of this research is very relevant to the question of matching type systems with relational data models [OB88, Oho88]. The path to take becomes much less clear however when we try to do the same thing with object-oriented databases and this is precisely the setting for the dilemma that we discuss in this paper.

3 The Problem and some Solutions

The major contribution of object-oriented languages is that class hierarchies support code sharing through the inheritance of methods. For example, if we define a class *POINT* with a method $MOVE(x, y)$ that displaces a point by co-ordinates x and y, we can define a subclass, *CIRCLE*, of *POINT* and expect that the same method, *MOVE*, can be applied to instances of the class *CIRCLE*. However this does not imply any obvious or natural relationship between the instances of *POINT* and the instances of *CIRCLE*. Even if there is such a relationship – we might implement a circle using an instance of *POINT* to describe the center – there could be many circles with the same center.

Unlike the inheritance hierarchies of object-oriented languages, object-oriented databases and semantic data models [HK87] impose a relationship on the instances of classes. For example, when we say an *EMPLOYEE* *isa* *PERSON*, we mean that, in a given database that the set of *EMPLOYEE* instances is a subset of the set of *PERSON* instances. As in object-oriented languages, we expect *EMPLOYEE* to be a subclass of *PERSON* in that every method (attribute) of *PERSON* is applicable to instances of *EMPLOYEE*. To clarify this idea, let us look at a sketch of a database query in which this subset assumption is made.

Example 1.

1. Obtain the set P of *PERSON*s in the database.
2. Perform some complicated restriction of P, e.g. find the subset of P whose *Age* is below the average *Age* of P.
3. Obtain the subset E of P of *EMPLOYEE*s in P.
4. Print out some information about E, e.g. print the names and ages of people in E with a given salary range.

The question is how to maintain or alternatively, simulate, such a set inclusion in a statically typed language.

3.1 Existing Approaches

Programming queries such as that in Example 1 are certainly possible in a dynamically typed language, and can easily be handled in object-oriented database languages. Of course this situation can be also handled in a statically typed system by maintaining disjoint sets of *EMPLOYEES* and *PERSONS* instances, each of which is uniformly typed, and at the same time by maintaining a "natural" embedding (injective map) from the set of *EMPLOYEES* to the set of *PERSONS* as a typed function in the language. This is what is done in the relational model through (seldom implemented) foreign key constraints. Implementing the query then involves a join at line 3 in order to find the employees who have a key that is in the relation containing the selected persons.

A slightly better approach would be to use disjoint union types to encode a "heterogeneous" set of all *PERSONS* and *EMPLOYEES* as a uniformly typed set and to define mappings on this set that yield the corresponding *PERSON* set and *EMPLOYEE* set. This is what the authors did in Machiavelli [OBBT89] through the use of "views". Although Machiavelli views provide a more natural way of deriving this embedding, they still require the user to construct them, and the underlying type specifications are rather complicated and somewhat reminiscent of Codasyl programming where the "database designer" has to implement a rather complicated schema and provide a number of subschemas (views) in order to make database programming palatable. An example may help to convey the idea. We start with `PersonObj` type which describes a person and has a variant to indicate whether a given object of that type also contains employee information. It should be noted that these type definitions are not essential in Machiavelli, they can be inferred from the data structures and function definitions involved.

```
type  PersonObj = ref([Name: string, Age: int;
                       Salary: <None: unit, Value: int>]);

type Person = [Name: string, Age: int, Id: PersonObj];

type Employee = [Name: string, Age: int, Salary: Integer, Id: PersonObj];
```

We then consider two data types that correspond to views of a `PersonObj`. These contain the fields we want for persons and employees together with a distinguished field `Id` that contains the `PersonObj` "object" itself.

```
fun PersonView(S) =
  select [Name=(!x).Name, Age=(!x).Age, Id=x]
  where x <- S
  with true;

fun EmployeeView(S) =
  select [Name=(!x).Name, Age=(!x).Age, (Salary=(!x).Salary as Value), Id=x]
  where x <- S
  with (case (!x).Salary of Value of _ => true, other => false);
```

The functions `PersonView` and `EmployeeView` are now defined. They apply to sets of of values of type

`PersonObj` to provide a set of values of type `Person` and a "subset" of values of type `Employee`. The query of example 1 now requires an application of `EmployeeView` at line 3 to select (and coerce) the appropriate subset of `Person` records to a set of `Employee` records. At the same time the type structure preserves the inclusion of the associated sets of `PersonObj` objects. Note that type declarations are not needed for these views; they are inferred. It is because of the special type inference that Machiavelli can maintain a statically typed approach to object-oriented programming.

Because of the way (derived from ML) that references are defined, it is possible to define set-theoretic operations on views. For example the *intersection* (which is also the *join* operator [BJO91]) is of type $\{\tau_1\} \times \{\tau_2\} \to \{\tau_1 \sqcup \tau_2\}$, where $\{\tau\}$ is the type of sets of values of type τ. Similar types can be given to other set operations, and membership and inclusion predicates can also be defined on views.

Thus, once views have been defined, programming in Machiavelli almost achieves our goal of having a static type system that will deal properly with sets of objects, and the solution to the query of example 1 looks the way we want it it to look. However the construction of complicated "catch-all" types like `PersonObj` and the definition of views – even though they could certainly be automatically generated from a semantic data model schema – is something we would prefer to avoid. It would be more natural if the schema were directly represented in the type system.

4 A Radical Solution: Heterogeneous Sets

Instead of using these somewhat ad-hoc encodings, we would like to develop a method for direct formulation of the query into a statically typed language. However, if we think of *PERSON* and *EMPLOYEE* as types then most, if not all, such languages will disallow the direct formulation for the following reason. In statically typed languages each value usually has a unique type and in any "bulk" type such as a list, array or set the type of each value in such a collection must be the same, i.e. we cannot put values of different types in the same list. This prohibits the line 3 of our query, because the elements of the set P are of type *PERSON* and cannot be of type *EMPLOYEE*.

To avoid this, (1) we could admit that values can have more than one type, or, (2) we can continue to insist on unique types, but then we must allow sets to contain values of different type. Both of these solutions require the use of heterogeneous sets.

4.1 Values have multiple types

A type system in which values can have more than one type has been suggested by Cardelli in [Car84b] and refined by Cardelli and Wegner[CW85] A "subtype relation" was introduced in order to represent *isa* hierarchies directly in the type system. For example, we can represent *PERSON* and *EMPLOYEE* by the following record types

$$\begin{aligned} PERSON &= [Name : string, Age : int] \\ EMPLOYEE &= [Name : string, Age : int, Sal : int] \end{aligned}$$

since the intended inclusion relation is captured by fact that $EMPLOYEE \leq PERSON$ holds in the type system. In these type systems, method sharing simply means type consistency of the desired applications. The intended type consistency is insured by the following typing rule (manifest in [CW85] and derivable in [Car84b]) :

$$\text{(sub)} \qquad \frac{e : \tau_1 \qquad \tau_1 \leq \tau_2}{e : \tau_2}$$

With such a rule, even simple objects such as records will have multiple types. If we add a set data type (a set type constructor) and the rule of set introduction

$$\text{(set)} \qquad \frac{e_1 : \tau, e_2 : \tau, ..., e_n : \tau}{\{e_1, e_2, ..., e_n\} : \{\tau\}}$$

we get a system that actually supports heterogeneous sets. For example, if $e_1 : \tau_1$ and $e_2 : \tau_2$ then e_1, e_2 also have the set of types $\overline{\tau_1} = \{\tau | \tau_1 \leq \tau\}$ and $\overline{\tau_2} = \{\tau | \tau_2 \leq \tau\}$. By applying the rule (set), the set-expression $\{e_1, e_2\}$ has any type $\{\tau\}$ such that $\tau \in \overline{\tau_1} \cap \overline{\tau_2}$. Furthermore, by using the property of the subtype relation that $\overline{\tau_1} \cap \overline{\tau_2} = \overline{\tau_1 \sqcap \tau_2}$, the type-checking algorithm of [Car84b] can be also extended to set expressions.

We may want to construct heterogeneous sets whose elements are heterogeneous sets themselves, so, for uniformity reasons, we might add

$$\text{(sub-set)} \qquad \frac{\tau_1 \leq \tau_2}{\{\tau_1\} \leq \{\tau_2\}}$$

This appears to be a promising approach to a a type system for object-oriented databases which supports code sharing and directly models set inclusion.

However, there seem to be certain problems with such a type system. We will first discuss what is usually called "loss of type information". This problem was first pointed out in [CW85] in the context of Cardelli's original type system, and *bounded quantification* was offered as a solution. We still have a couple of comments to make about this.

To review what loss of type information is, consider:

$$F1 \equiv \lambda x : [Name : string].\ x$$

This function does not "behave" like an identity on all the objects to which it is applicable (i.e., for which the application type-checks). For example, $E1 \equiv F1\,[Name = "Joe\ W\ Doe", Age = 21]$ type-checks, but $E1 \cdot Age$ does not, and, in this sense, $E1$ is not interchangeable with $[Name = "Joe\ W\ Doe", Age = 21]$. The solution proposed in [CW85] introduces bounded quantification, which is a generalized form of polymorphic type abstraction [Rey74]. In the new language, $F1$ is not banned but in order to implement the identity function that $F1$ does not quite capture we would use

$$F1' \equiv \Lambda t \leq [Name : string].\ \lambda x : t.\ x$$

$F1'$ is now a function which must first be applied to a type. What this accomplishes is that, $(F1'\,[Name : string, Age : int])\,[Name = "Joe\ W\ Doe", Age = 21]$ type-checks in exactly the same contexts as $[Name =$

"$Joe\ W\ Doe$", $Age = 21$] does. Have we provided a "good" replacement for $F1$? Clearly, this replacement cannot be $(F1'\,[Name : string, Age : int])$ because it is not applicable to some objects to which $F1$ is. In some sense, $F1'$ *is* the replacement, but this point of view must be taken with caution since $(F1'\,[Name : string])\,[Name = "Joe\ W\ Doe", Age = 21]$ also type-checks, and "behaves" just as badly as $E1$ above.

Even if we accept this point of view, corrections by bounded quantification cannot be generalized naively from the previous example. Consider the "bad" function [OB88]:

$$F2 \equiv \lambda x : [Name : [Fn : string, Ln : string]].\ x \cdot Name$$

Because subtyping between record types is defined hereditarily

$$\text{(sub-record)} \qquad \frac{\sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_n \leq \tau_n}{[l_1 : \sigma_1, \ldots, l_n : \sigma_n] \leq [l_1 : \tau_1, \ldots, l_m : \tau_m]} \quad m \leq n$$

we have the following loss of type information problem: $E2 \equiv F2\,[Name = [Fn = "Joe", Mi = "W", Ln = "Doe"]]$ type-checks, but does not behave like $[Fn = "Joe", Mi = "W", Ln = "Doe"]$ since $(E2 \cdot Name) \cdot Mi$ does not type-check. This problem is *not* corrected by

$$F2' \equiv \Lambda t \leq [Name : [Fn : string, Ln : string]].\ \lambda x : t.\ x \cdot Name$$

since $E2' \equiv (F2'\,[Name : [Fn : string, Mi : string, Ln : string]])\,[Name = [Fn = "Joe", Mi = "W", Ln = "Doe"]]$ type-checks, but $(E2' \cdot Name) \cdot Mi$ does not. This is because the smallest type that can be derived for $F2'$ is

$$\forall t \leq [Name : [Fn : string, Ln : string]].\ t \to [Fn : string, Ln : string]\ .$$

Instead, a "good" replacement for $F2$ is

$$F2'' \equiv \Lambda t_1 \leq [Fn : string, Ln : string].\ \Lambda t_2 \leq [Name : t_1].\ \lambda x : t_2.\ x \cdot Name$$

since $((F2''\,[Fn : string, Mi : string, Ln : string]\,[Name : [Fn : string, Mi : string, Ln : string]])\,[Name = [Fn = "Joe", Mi = "W", Ln = "Doe"]])$ type-checks in exactly the same contexts as $[Fn = "Joe", Mi = "W", Ln = "Doe"]$ does.

A precise analysis of the "loss of type information" phenomenon is certainly desirable. Meanwhile, we can offer a comment that may contribute towards understanding part of the last example. Using the rule (*sub*) to type-check the terms containing field selection $e \cdot l$ seems to fail reflect the precise operational behavior of this program construction, which is

$$[\ldots, l = e, \ldots].l \Rightarrow e$$

A typing rule that would fit this is:

$$\text{(dot)} \qquad \frac{e : \tau_1}{e \cdot l : \tau_2} \qquad \text{where } \tau_1 \text{ is a record type containing } l : \tau_2$$

The associated condition coincides with the subtype relation $\tau_1 \leq [l : \tau_2]$ *only if* τ_2 has no proper subtype. However, if τ_2 is a type that has proper subtypes such as record types, then the condition associated with the typing rule (dot) is strictly stronger than the subtype relation. This may help explaining why $F2$ exhibits loss of type information, but, for example, $\lambda x : [Name : string].\ x \cdot Name$ does not. The rule (dot) is used in Machiavelli in its type inference algorithm.

The presence of the rule (*sub*) also raises some problems with primitive operations that are important for database programming such as join and equality. Here we consider the treatment of equality. In the presence of (sub), it makes sense to type-check $e_1 = e_2$ whenever $e_1 : \tau_1$ and $e_2 : \tau_2$ such that there exists a τ_3 with $\tau_1 \leq \tau_3$ and $\tau_2 \leq \tau_3$ (which will always be the case when τ_1 and τ_2 are record types!). Adopting the static type-checking philosophy, we must give an operational meaning to, say, $[a = "Nick", b = 1] = [a = "Nick", c = false]$. Amber[Car84a], which has the (sub) rule, takes a safe, but rather weak position here: equality is implemented by identity between run-time objects in store (like the *eq* predicate in Lisp). In this view, $[a = "Nick", b = 1]$ and $[a = "Nick", c = false]$ are not equal, but, in fact, $[a = "Nick"] = [a = "Nick"]$ presumably also evaluates to false.

It can be argued that a stronger view of equality, one that takes into account the type structure of the language, is desirable. An example of such "structural" equality is the primitive = in Standard ML. It is not at all clear how to extend structural equality to cases like $[a = "Nick", b = 1] = [a = "Nick", c = false]$. One possibility is to raise a run-time exception, arguing that this situation resembles division by 0, that is, it's really a type error, but there is no useful decidable static type-checking system that includes (sub) and that would catch it.

Another approach would be to tag equality with types and to have the following type-checking rule:

$$\text{(eq)} \qquad \frac{e_1 : \tau_1 \quad \tau_1 \leq \tau \quad e_2 : \tau_2 \quad \tau_2 \leq \tau}{e_1 =_\tau e_2 : bool}$$

$e_1 =_\tau e_2$ is then implemented as a test for equality of the "τ-parts" of e_1 and e_2. $[a = "Nick", b = 1]$ and $[a = "Nick", c = false]$ are then $[a : string]$-equal (and for that matter, $[\,]$-equal). Moreover, there is no type τ such that these two objects are τ-different! This approach, which may seem peculiar at first glance, is supported, and in fact suggested, by the coercions interpretation of (sub) presented in [BTBO89]. However, the usefulness of this form of equality needs more investigation. This equality also requires some type information from τ (the nested field structure) to be kept in order to to evaluate $=_\tau$ at run-time.

In any case, it appears that more work is needed on these kind of type systems. It is possible that the techniques described in [Car88a] can be used to solve some of these problems. Studies of these type systems offer some exciting challenges, and there have been several semantic studies (see [BTBO89] and references therein). Some syntactic studies seem highly desirable too. For example, we expect that type-checking for the system with bounded quantification in [CW85] is decidable, but, while we can imagine what the algorithm should be, we know of no proof for its termination.

4.2 Types are unique, but sets are heterogeneous

Recently, there has been another approach to subtyping which is closely related to type inference. In a method originally suggested by [Wan87] and elaborated in various ways in [Sta88, JM88, OB88, Rem89], the fact that a function such as $\lambda x.\, x \cdot Name$ can be applied to records of various types is discovered through type inference. Moreover in [OB88] it is demonstrated that this process can be used to infer types for a set of database operations that includes, for example, the operations of the relational algebra. In these approaches objects have a unique type and, following our argument, disallow sets to contain values of different

types if typing rule for set is homogeneous like the rule (set) above. The second possibility of supporting object-oriented database might therefore be to allow heterogeneous set in these type system:

$$\{e_1, e_2, ..., e_n\} : \{\tau_1, \tau_2, ..., \tau_n\}$$

In order to build a type system based on this idea, we need to decide the meaning of the typing judgement $\{e_1, e_2, ..., e_n\} : \{\tau_1, \tau_2, ..., \tau_m\}$. There are at least the following three possiblities:

1. for each e_i there is some τ_j such that $e_i : \tau_j$,
2. for each τ_i there is some e_j such that $e_j : \tau_i$,
3. the conjunction of the two.

Unfortunately none of these provide a satisfactory interpretation. The first interpretation implies the set expressions have more than one type yielding the same problem as a subtype based system. The other two interpretations do not support necessary operations on sets including intersection and difference. For example, under either of the two interpretations, $S_1 : \{\tau_1, ..., \tau_n\}$ and $S_2 : \{\tau_1, ..\tau_n\}$ does not imply $S_1 \cap S_2 : \{\tau_1, ..., \tau_n\}$. The second interpretation is also unsafe when combined with function application.

The only alternative we are left with is to introduce some notion of *partial type information* for sets. For example, we would like to specify that every member of a set has a $Name : string$ field. One way to specify partial type information is to specify a set of possible types. In the theory of types, sets of types are sometimes called *kinds* [Mac79] and are treated as themselves objects. The special set constructor can then be characterized as an operator takes a kind and returns a type. In particular, the following kinds representing various sets of record types seem particularly useful in object-oriented databases:

$$\kappa ::= K(\tau) \mid (l : \kappa, \ldots, l : \kappa)$$

where $K(\tau)$ is the kind correspond to the singleton set of τ and $(l_1 : \kappa_1, \ldots, l_n : \kappa_n)$ denote the set of all record types containing at least all the fields $l_1, \ldots, l_n$ of types specified by respective kinds. This relation can be easily formalized by kinding rule, which also induce a partial order on kinds.

The set of types can be extended by partial types:

$$\tau ::= T(\kappa) \mid \ldots$$

where $T(\kappa)$ is the type with partial type information represented by the kind κ. Since our objective is to integrate heterogeneous sets, the following introduction rule seems the only necessary introduction rule for partial types.

$$\text{(HSET)} \qquad \frac{e : \sigma}{\{e\} : \{T(K(\sigma))\}}$$

Possible elimination rules are:

$$\text{(COERCE)} \qquad \frac{e : T(K(\sigma))}{e : \sigma}$$

$$\text{(DOT)} \qquad \frac{e : T((\ldots, l : \kappa, \ldots))}{e \cdot l : T(\kappa)}$$

For example

$$\{[Name = "Joe", Age = 21], [Name = "Helen", Sal = 30k]\}\{T((Name : string))\}$$

Different form the standard set type $S : \{[Name : string]\}$, applicable functions are limited to those whose only interaction with the members of S is to select the $Name$ field. Union and intersection can be generalied to these partially specified set types.

$$\text{(UNION)} \qquad \frac{e_1 : \{T(\kappa_1)\} \qquad e_2 : \{T(\kappa_2)\}}{union(e_1, e_2) : \{T(\kappa_1 \sqcup \kappa_2)\}}$$

$$\text{(INTERSECT)} \quad \frac{e_1 : \{T(\kappa_1)\} \qquad e_2 : \{T(\kappa_2)\}}{intersection(e_1, e_2) : \{T(\kappa_1 \sqcap \kappa_2)\}}$$

where $\kappa_1 \sqcup \kappa_2$ and $\kappa_1 \sqcap \kappa_2$ denote respectively the least upper bound and greatest lower bound of kinds under the set inclusion ordering. It seems not hard to integrate this form of condition to a static type inference system such as [OB88] and obtain a type safe integration of heterogeneous sets in a type system allowing object-oriented programming via type inference. This is appealing because the necessary conditions to devlop a type inference algorithm seems to be very similar to those required for the relational algebra and implemented in Machiavelli. However, it is not yet clear what kind of partial type information is useful and what kind of operations are necessary and sufficient to support object-oriented database. The above examples indicate that the subtype relationship in [Car84b] may be more appropriate at the level of kinds - where it simply indicates set inclusion.

5 Conclusions

As promised, our conclusions are inconclusive. We have attempted to describe various obstacles to combining object-oriented databases and semantic data models with a static type system. In particular, if we accept the fact that values do not have unique types, we seem to be led to type systems with certain undesirable behavior, specifically the "loss of information" problem. However, we believe that these systems merit further study and that there may be ways of fixing this problem. The other possibility is to have a system in which each value has a unique type, but we are now led to introduce heterogeneous sets which cannot be regarded as regular values in the language because they do not have a conventional type. Any type information about the set can only be regarded as partial information about the types of the object in that set. The status of this partial type information remains to be clarified.

A related issue arises in the problem of type definition. In the second approach to subtyping, we exploited type inference in a fashion that made type definition unnecessary except, perhaps, because the programmer wants to put some constraints on the types that the system can infer. However, in database programming we definitely need these constraints. The problem arises because, in database programming, we cannot predict in advance what programs will be written against a particular database. To ensure the type safety of such

programs some form of type definition is essential to ensure that the database is updated in a fashion that is consistent with its intended use. Of course, the user who writes an application against the data base does not necessarily need the whole type declaration (schema) for the database; all that is needed is a guarantee that the inferred type of a program is consistent with a some part of the database schema. Again some concept of partial type information may be relevant.

References

[AB87] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, June 1987.

[BJO91] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91(1):23–56, 1991.

[BTBG88] V. Breazu-Tannen, P. Buneman, and C.A. Gunter. Flexible type systems for the rapid development of reliable software. In J. E. Gaffney, editor, *Productivity: Progress, Prospects and Payoff*. Association for Computing Machinery, June 1988.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Static type-checking in object-oriented databases. *IEEE Data Engineering, Special Issue on Database Programming Languages*, 12(3):5–12, 1989.

[Car84a] L. Cardelli. Amber. Technical Memorandum TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[Car84b] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*. Springer-Verlag, 1984.

[Car88a] L. Cardelli. Quest. Technical report, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301, 1988.

[Car88b] L. Cardelli. Structural subtyping and the notion of power types. In *Proc. ACM Symposium on Principles of Programming Languages*, San Diedo, California, January 1988.

[Car88c] L. Cardelli. Types for data-oriented languages (overview). In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proceedings of the International Conference on Extended Database Technology, Lecture Notes in Computer Science 303*, pages 1–15, Venice, Italy, March 1988. Springer-Verlag.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[FM88] Y-C. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of ESOP '88*, pages 94–114, 1988. Springer LNCS 300.

[GB80] I. P. Goldstein and D. G. Bobrow. Extending object oriented programming in smalltalk. In *Proceedings of the 1980 Lisp Conference*, pages 75–81, August 1980.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *Computing Surveys*, 19(3), September 1987.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML (version 2). LFCS Report Series ECS-LFCS-88-62, Department of Computer Science, University of Edinburgh, August 1988.

[Hul87] R. Hull. *Databases*, chapter 5 A survey of theoretical research on typed complex database objects, pages 193–253. Academic Press, 1987.

[JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[Mac79] N. Maccracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse University, 1979.

[Mit84] J.C. Mitchell. Coercion and type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. the ACM SIGMOD conference*, pages 46–57, Portland, Oregon, May – June 1989.

[Oho88] A. Ohori. Semantics of types for database objects. In *Proc. International Conference on Database Theory, Lecture Notes in Computer Science 326*, pages 239–251, Bruges, Belgium, August 1988. Extended version in a special issue of *Theoretical Computer Science* dedicated to 2^{nd} ICDT.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 242–249, 1989.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16. Springer-Verlag, 1985.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987. a corrigendum in *Proc. Symposium on Logic in Computer Science*, 1988.