Static Type-checking in Object-Oriented Databases[*]

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 Introduction

If a precise definition of object-oriented programming languages is elusive, the confusion surrounding *object-oriented databases* is even greater. Rather than attempt to give a comprehensive definition of the subject we shall concentrate on a few properties of object-oriented databases that we believe to be of central importance. We want to show that these properties can be concisely captured in a language that has a more-or-less conventional type system for the representation of data, and that achieves its "object-orientedness" by exploiting type inference. The advantage of this approach is that programs are statically checked for type correctness without the programmer having to declare types. By doing this we believe we can eliminate a major source of errors in programming on databases – type errors, which proliferate as the complexity of the database increases. In most object-oriented database systems, type errors are not caught until something goes wrong at run-time, often with disastrous consequences.

Let us briefly discuss three properties of object oriented languages and databases that will figure in our presentation. There are of course other features, but we shall defer a discussion of these until the end of this paper.

Method inheritance. This features in all object-oriented languages and describes the use of a programmer-defined hierachy to specify code-sharing. Code defined for some class is applicable to objects in any subclass of that class. For example, a programmer could define a class *POINT* with an associated method $Move(x, y)$ that displaces a point by co-ordinates x and y. Subsequently, a subclass *CIRCLE* of point may be defined, which means that the method *Move* is also applicable to objects of class *CIRCLE*.

Object identity. In the precursors to object-oriented languages [DN66], which were used for simulation, an object could represent a "real-world" object. Since real world objects can change state, the correspondence

[*] This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634. This research was done during the third author was on leave from Oki Electric Industry. Atsushi Ohori's current address: Oki Electric Industry, Kansai Laboratory, Crystal Tower, 1-2-27 Shiromi, Chuo-ku, Osaka 540, JAPAN

Published in **IEEE Data Engineering Bulletin**, 12(3):5–12, 1989. Special issue on Database Programming Languages.

between a program object and a real-world object was maintained by endowing the program object with "identity" - a property that remained invariant over time and served to distinguish it from all other objects. Since the purpose of object-oriented databases is also, presumably, to represent the real world, we would expect it to figure in our discussion.

Extents. In any database one needs to maintain large collections – lists or sets, for example – of objects with certain common properties. Typically, a database might contain *EMPLOYEES* and *DEPARTMENTS*, each describing a collection of values with some common properties. For example, we would expect *SALARY* information to be available for each member of of *EMPLOYEES*. The need to deal efficiently with extents is one of the distinguishing features of object-oriented databases; and the connection between extents and classes will be one of the main foci of our discussion.

On initial examination of these properties, it is tempting to tie extents to classes, but this immediately creates problems. In the case of *POINT* and *CIRCLE* there is no obvious relationship between the objects of these two classes, and 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. On the other hand, when we say (in the jargon of semantic networks and data models [HK87]) that *EMPLOYEE isa PERSON*, we mean that, in a given database, the set of *EMPLOYEE* instances is a subset of the set of *PERSON* instances.

There is an immediate contradiction if we think of *EMPLOYEES* as the set of all objects of class *EMPLOYEE* and similarly for *PERSON*. For if an object is in class *EMPLOYEE*, it cannot be in class *PERSON*. Therefore *EMPLOYEES* cannot be a subset of the set of instances of *PERSON*. We probably want to consider the set of persons to be the union of the *PERSON* and *EMPLOYEE* objects, but this does not bode well for static type-checking because this set is now heterogeneous, and it is not clear what type to give it.

In the following sections, we shall outline an approach to this problem that relies heavily on type inference for record types. The ideas are embodied in an experimental language Machiavelli [OBB89] that has been implemented at the University of Pennsylvania.

2 Type Inference and Inheritance

To see how we can derive methods through type inference, consider a function which takes a set of records (i.e. a relation) with Name and Salary information and returns the set of all Name values for which the corresponding Salary values are over 100K. For example, applied to the relation

```
{[Name = "Joe", Salary = 22340],
 [Name = "Fred", Salary = 123456],
 [Name = "Helen", Salary = 132000]}
```

this function should yield the set {`"Fred", "Helen"`}. Such a function is written in Machiavelli (whose syntax mostly follows that of ML [HMT88]) as follows

```
fun Wealthy(S) = select x.Name
                 where x <- S
                 with x.Salary > 100000;
```

Although no data types are mentioned in the code, Machiavelli *infers* the type information

```
Wealthy: {[("a) Name:"b,Salary:int]} -> {"b}
```

by which it means that `Wealthy` is a function that takes a homogeneous set of records, each of type `[("a) Name : "b, Salary : int]`, and returns a homogeneous set of values of type `"b`, where `("a)` and `"b` are *type variables*. `"b` represents an arbitrary type on which equality is defined. `("a)` represents an arbitrary extension to the record structure that does not contain `Name` and `Salary` fields; this is superficially similar to the "row variables" in [Wan87]. `"b` and `("a)` can be *instantiated* by any type and record extension satisfying the above conditions. Consequently, Machiavelli will allow `Wealthy` to be applied, for example, to relations of type

```
{[Name: string, Age:int, Salary: int]}
```

and also to relations of type

```
{[Name: [First: string, Last: string],
  Weight: int, Salary:int]}.
```

The function `Wealthy` is *polymorphic* with respect to the type `"b` of the values in the Name field (as in ML) but is also polymorphic with respect to extensions `("a)` to the record type `[Name:"b ,Salary: int]`. In this second form of polymorphism, `Wealthy` can be thought of as a "method" in the sense of object-oriented programming languages where methods associated with a class may be inherited by a subclass, and thus applied to objects of that subclass. In other words, we can think of `{[Name:"b ,Salary: int]}` as the description of an inferred "class" for which `Wealthy` is an applicable method.

For the purposes of finding a typed approach to object-oriented programming, Machiavelli's type system has similar goals to the systems proposed by Cardelli and Wegner [Car84, CW85]. However, there are important technical differences, the most important of which is that database values have *unique types* in Machiavelli while they can have multiple types in [Car84]. Based on the idea suggested in [Wan87], Machiavelli achieves the same goals of representing objects and inheritance (see also [Sta88, JM88] for related studies). These differences allow Machiavelli to overcome certain anomalies (see [OB88], which also gives details of the underlying type inference system).

3 Representing Objects and Extents

The example we have just presented is intented to illustrate how Machiavelli can infer types in a function defined over a set of records (a relation). In fact, the `select ... where ... with ...` is a simple syntactic sugaring of a combination of a small number of basic polymorphic operation on sets and records, which provide a type inference system for an extended relational algebra. The reader is referred to [OBB89] for details, for space does not allow us to describe them here. Instead we turn to the problem that we posed in the introduction of combining the two views of an inheritance hierarchy: as a structure that describes the inheritance of methods and as a structure that describes containment between sets of extents.

Suppose we have two sets, E a set of objects of type *Employee* and S a set of objects of type*Student* and we wish to take the intersection. We would expect objects in $E \cap S$ to inherit the methods of both *Student* and *Employee*. But note that our intersection is rather strange because it operates on sets of different types. In fact, we want its type to be $\{\tau_1\} \times \{\tau 2\} \rightarrow \{\tau_1 \sqcap \tau_2\}$. In particular, if τ_1 and τ_2 are record types, then the

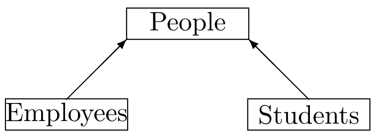

Figure 1: A Simple Class Structure

operation we want looks very much like the natural join since it takes the union of the methods (attribute names) on something that looks like the intersection of the rows. But without object identity, it is not clear exactly what the intersection of rows means in this case. Our first task is to introduce some notion of object identity.

We claim that reference types in ML, upon which Machiavelli is based, capture the essence of object identity. For example, if we set up a reference to a record type as follows

```
val d = ref([Dname="Sales", Building=45]);
```

and from this we define two employee records

```
val emp1 = ref([Name = "Jones", Department = d]);
val emp2 = ref([Name = "Smith", Department = d]);
```

then an update to the building of the department as seen from `emp1`

```
let val d = (!emp1).Department
in d:=modify(!d, Building, 67)
end;
```

will be reflected in the department as seen from `emp2`. Also, two references are equal only if they are the result of the same invocation of the function `ref` which creates references. For example, `ref(3) = ref(3)` is *false*, the two applications of `ref` generate different (unequal) references. Thus different references can never be confused even if they refer to the same value.

Now suppose we wish to represent the hierarchy shown in figure 1. Again, note that we want the arrows not only represent inheritance of properties but also actual set inclusions

To do this we start by considering a *PersonObj* type as being sufficiently rich to contain all the possible states of a person; variant types are used, for example, to indicate whether or not a person is an employee:

```
PersonObj = ref([Name:  string, Salary :  <None:  unit, Value:  int>,
                 Advisor :  <None:  unit, Value:  PersonObj>]
```

This is not a type declaration in Machiavelli (type declarations are not in general needed). *PeronObj* is simply a name we shall give to a type that a particular data structure may or may not posses.

Now *PersonObj* is a rather complicated type, and its relationship to the hierarchy is not immediately clear. What we want to deal with are some types that directly provide the information we need:

Person = `[Name: string, Id: PersonObj];`
Student = `[Name: string, Advisor: PersonObj, Id: PersonObj]`
Employee = `[Name: string, Salary: Integer, Id: PersonObj]`

Again these are just convenient shorthands that we shall use in some of the examples that follow. The type *Person*, *Employee* and *Student* directly provide the information we want for these classes, but they also contain a distinguished field, the `Id` field, that retains the person object from which each record is derived.

Now suppose we are provided with a set of person objects, i.e. a set of type {`PersonObj`}, we can write in Machiavelli, functions that "reveal" some of the information in this set. We call such a function a *view*.

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

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

fun StudentView(S) =
       select [Name=(!x).Name, (Advisor=(!x).Advisor as Value), Id=x]
       where x <- S
       case (!x).Advisor of
           Value of _ => true,
           other => h (case (!x).Course of Value of _ => true, other => false);
```

The types inferred for these functions will be quite general, but the following are the instances that are important to us in the context of this example.

`PersonView :` $\{PersonObj\}$ `->` $\{Person\}$
`EmployeeView :` $\{PersonObj\}$ `->` $\{Employee\}$
`StudentView :` $\{PersonObj\}$ `->` $\{Student\}$

We are now in a position to write a function that, given a set of persons, extract those that are both students and employees:

```
       fun supported_students(S) = join(StudentView(S),EmployeeView(S);
```

In the definition of `supported_students`, the join of two views models both the intersection of the two classes and the inheritance of methods. If τ, σ are types of classes, then $\tau \leq \sigma$ implies that `Project`$(View_\sigma(S), \tau) \subseteq$

$View_\tau(S)$) where $View_\tau$ and $View_\sigma$ denote the corresponding viewing functions on classes τ and σ. This property guarantees that the join of two views corresponds to the intersection of the two. The property of the ordering on types and Machiavelli's polymorphism also supports the inheritance of methods. Thus the methods applicable to `StudentView(S)` and `EmployeeView(S)` are automatically inherited by Machiavelli's type inference mechanism and are applicable to `supported_students(S)`. As an example of inheritance of methods, the function `Wealthy`, as defined in the introduction, has type `{[("a) Name:"b,Salary:int]} -> {"b}`, which is applicable to `EmployeeView(S)`, is also applicable to `supported_students(S)`.

Dual to the join which corresponds to the intersection of classes, the union of classes can be also represented in Machiavelli. The primitive operation `unionc` is a generalization of the union defined in connection with `hom` to the operate on type $\{\delta_1\} * \{\delta_2\}$ for all description types δ_1, δ_2 such that $\delta_1 \sqcap \delta_2$ exists. Let s_1, s_2 be two sets having types $\{\delta_1\}, \{\delta_2\}$ respectively. Then $\texttt{union}(s_1, s_2)$ satisfies the following equation:

$$\begin{aligned}&\texttt{union}(s_1, s_2) = \\ &\quad \texttt{project}(s_1, \delta_1 \sqcap \delta_2) \cup \texttt{project}(s_2, \delta_1 \sqcap \delta_2)\end{aligned}$$

which is reduced to the standard set-theoretic union when $\delta_1 = \delta_2$. This operation can be used to give a union of classes of different type. For example, `union(StudentView(person), EmployeeView(person))` correspond to the union of students and employees. On such a set, one can only safely apply methods that are defined both on students and employees. As with join, this constraint is automatically maintained by Machiavelli's type system because the result type is {`Person`}.

In addition one can easily define the "membership" operation on classes of disparate type:

```
   fun member(x,S) = join({x},S) <> {}
```

`member(x,S) = true` iff there is some member of s of `S` such that `x` and s have a common identity. In this fashion it is possible to extend a large catalog of set-theoretic operations to classes.

4 Some Alternative Approaches

While the solution presented above provides a reasonable reconciliation of two forms (method sharing and subset) of inheritance, there may be alternaive approaches. One problem with our solution is that it requires the explicit construction of views, and while this follows naturally – perhaps automatically – from the class hierarchy, having to use som *ad hoc* encoding is not entirely satsifactory. Consider the following query

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.

In Machiavelli, at line 3, one will have to perform an explicit coercion from *PERSON* to *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 types. Both of these solutions require the use of heterogeneous sets. These are discussed in more detail in [BBO89]. We limit ourselves to a few remarks here. A type system in which values can have more than one type has been suggested by Cardelli in [Car84] 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

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

since the intended inclusion relation is captured by the fact that $\mathit{EMPLOYEE} \leq \mathit{PERSON}$ holds in the type system. In these type systems, method sharing simply means type consistency of the desired applications. The intended type consistency is ensured by the following typing rule (manifest in [CW85] and derivable in [Car84]) :

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

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 heterogenous 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 typechecking algorithm of [Car84] can be also extended to set expressions.

However there are certain drawbacks to this system. It suffers from the problem called "loss of type information", i.e. there are certain terms that one would reasonably expect to type-check, but are not typable under the given typing rules. This anomaly could be avoided by performing appropriate type abstractions and type applications. However, giving an appropriate type to the simplest of functions is not always trtivial. For example, the function that extracts the *Name* field from a record is implemented by the following polymorphic term:

$$\mathit{pname} \equiv \Lambda t_2 \Lambda t_1 \leq [\mathit{Name} : t_2] \lambda x : t_1 . x \cdot \mathit{Name}$$

which must be applied to the appropriate types before it can be evaluated. This is practically rather cumbersome. Worse yet, there seems no uniform way to find appropriate type abstractions and type applications to avoid the anomaly.

Another possibility of supporting object-oriented database is to allow heterogeneous sets and to keep *partial* type information about such sets. For example it should be possible to assert for a set of records that each record contains a $Name : string$ field, and to use such an assertion to type-check an application that requires access to the name field of each record in the set even though those records may be of differing type. The detailed syntax of such a system have yet to be worked out, but some proposals are given in [BBO89].

References

[BBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can Object-Oriented Databases be Statically Typed? In *Proc. 2^{nd} International Workshop on Database Programming Languages*, Morgan Kaufmann Publishers, Gleneden Beach, Oregon, June 1989.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer-Verlag, 1984.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[DN66] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[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.

[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.

[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.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli – a Polymorphis Language with Static Type Inference. In *Proceedings of the ACM SIGMOD conference*, pages 46–57, May – June 1989.

[Sta88] R. Stansifer. Type Inference with Subtypes. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Wan87] M. Wand. Complete Type Inference for Simple Objects. In *Proceedings of the Second Anual Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.