Data Structures and Data Types for Object-Oriented Databases*

Val Breazu-Tannen, Peter Buneman and Atsushi Ohori†

1 Introduction

The possibility of finding a static type system for object-oriented programming languages was initiated by Cardelli [Car88, CW85] who showed that it is possible to express the polymorphic nature of functions such as

```
fun age(x) = thisyear - x.year_of_birth
```

which may be regarded as a method of the "class" of record values that contain a numeric `age` field. It is possible both to integrate this form of record polymorphism with the parametric (universal) polymorphism and also to express a number of object-oriented programming paradigms by combining higher-order functions with field selection. Since then a number of alternative schemes [Wan87, Sta88, OB88, JM88, Rem89, HP91] have been developed that include the possibility of type inference and the use of abstract types [OB89]. The extent to which these typing schemes give a satisfactory account of all aspects of object-oriented languages remains an open question, and it may therefore be premature to complicate the picture by introducing database concepts. Nevertheless, if we are to treat databases of any kind (object- oriented or otherwise) properly in typed programming languages there are certain issues that must be resolved, and it is these that we shall briefly investigate in this paper.

Unlike the languages associated with the relational data model which have simple and more or less coincident operational and denotational semantics, the authors know of no equivalent formulation of an object-oriented data model. While several papers, e.g. [ABD$^+$89] describe certain *desiderata* of object-oriented databases, a consensus has yet to emerge on a formalism for an object-oriented data model, nor is there yet an adequate explanation of what is fundamentally new about such a model. The issues we discuss in this paper are relevant to object-oriented databases because they are of general concern in languages [Sch77, ABC$^+$83, ACO85] that integrate database structures with their type systems, and object-oriented databases surely fall into this category.

We shall be mainly concerned with operations on records and some "bulk" data type such as sets. Any database programming language [AB87] must surely be capable of expressing a function such as

```
fun wealthy(S) = select x.Name
                 from x <- S
                 where x.Sal >= 100,000
```

The syntax here is taken from Machiavelli [OBBT89], but very similar definitions are to be found in object-oriented languages such as O_2 [LRV88]. We would like a type system to express exactly what is required of the argument `S` in order for the function `wealthy` to be well defined. `S` contains *records* (perhaps objects) with appropriate properties. `S` must be a set (or some other bulk type such as a bag or list.) Finally we must allow that in some databases `S` may be *heterogeneous*: the individual records may not all have the same structure. These three demands on a programming language are the issues we discuss in this paper.

* Published in **IEEE Data Engineering bulletin, Spetial issue on Theoretical Foundations of Object-Oriented Database Systems** 14(2):23–27, June 1991

† Authors addresses: Buneman and Breazu-Tannen, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389, USA; Ohori, Kansai Laboratory, OKI Electric Industry, Crystal Tower, 1-2-27 Shiromi, Chuo-ku, Osaka 540, Japan.

Breazu-Tannen was partially supported by grants ONR NOOO-14-88-K-0634,NSF CCR-90-57570. Buneman was partially supported by grants ONR NOOO-14-88-K- 0634,NSF IRI-86-10617 and by a UK SERC visiting fellowship at Imperial College, London.

2 Operations on records

In [OBBT89] the function `wealthy` is given the type

```
{[('a) Name:'b, Sal:num]} -> {'b}
```

The `'a` and `'b` are *type variables* and may, subject to restrictions discussed below, be instantiated by any type. The notation `{'b}` describes the type of sets of values of type `'b`, and the notation `[('a) Name:'b, Sal:num]` describes the restriction that an instance of `'a` must be a record type that contains the fields `Name :` σ and `Sal:num` where σ is any instance of `'b`. For example

```
{[Name:string, Sal:num, Weight:num]} -> {string}
{[Name:[First:string, Last:string], Sal:num]} -> {[First:string, Last:string]}
```

are legal instantiations of the type of `wealthy`.

Using such a syntax it is not only possible to express the exact polymorphic type of a function like `wealthy`; it is possible to *infer* a type by means of an extension of ML's type inference system. If the only operations on records were record formation and field selection, the necessary techniques are now well established (in fact the approaches given in [Wan87, Sta88, OB88, JM88, Rem89] would agree.) The differences arise when we add operations that extend or combine records, and this is where databases place an unusual demand on the type system. An operation common in databases is to *join* two records on consistent information. For example `[Name='Joe', Age=21]` and `[Name='Joe', Sal=30,000]` join to form `[Name='Joe', Age=21, Sal=30,000]`. On the other hand there is no type that can be given to the join of `[Id=1234]` and `[Id='A123']`. This join can be extended to sets of records i.e. relations, and in fact to arbitrary structures on which equality is provided, to define the *natural join* of complex objects [Oho90b]. It is arguable that natural join is needed in a database programming language, but even if it is not, a very similar typing rule is needed for the intersection of heterogeneous sets (see below).

There is a well-known result [Mil78] that underlies the polymorphic type system of ML that every expression has a *principal* type scheme, i.e. every possible ground type for an expression can be obtained by instantiating the type variables of its principal type. If we add the typing rules for record formation and field selection:

$$(\textsc{record})\quad \frac{\mathcal{A}\rhd e_1 : \tau_1,\ \ldots,\ \mathcal{A}\rhd e_n : \tau_n}{\mathcal{A}\rhd [l_1{=}e_1,\ldots,l_n{=}e_n] : [l_1:\tau_1,\ldots,l_n:\tau_n]} \qquad (\textsc{dot})\quad \frac{\mathcal{A}\rhd e : [\ldots,l:\tau,\ldots]}{\mathcal{A}\rhd e.l : \tau}$$

we retain the principal typing property [Oho90a]. However, the rule for join is unusual in that it can only be used provided a "side condition" is satisfied:

$$(\textsc{join})\quad \frac{\mathcal{A}\rhd e_1 : \delta_1 \qquad \mathcal{A}\rhd e_2 : \delta_2}{\mathcal{A}\rhd \mathbf{join}(e_1,e_2) : \delta} \qquad \text{if } \delta = \delta_1 \sqcup \delta_2$$

The requirement that any ground type also satisfy the side conditions means that we need to relax the notion of the principal typing property to include these conditions; nevertheless it is still decidable whether a given expression has a type, and by suitably delaying the checks for satisfaction, the process of type inference can be made efficient and to operate interactively.

3 Operations on sets

The operations of the relational algebra suggest one way in which operations on sets may be added to a programming language. While these are adequate for a large number of database applications, there are a number of useful operations, such as transitive closure of a binary relation, that cannot be expressed with the relational algebra alone. Moreover, there is no way of expressing the cardinality, sum, or other aggregate operations on a set. Relational query languages provide these as special operations, but there is no general way to construct new aggregate operations.

The problem is this: the relational algebra provides us with an adequate set of operations for mapping sets of tuples (records) into sets of tuples, but provides us with no way of moving outside this domain; we cannot expect the relational algebra to produce a set of sets or an integer. One could get rounds this by adding a *choose* operator, which picks arbitrarily an element of a set, and using general recursion to program functions mapping sets to other types. However, *choose* introduces a nondeterministic

semantics, and makes it difficult to ensure that our programs are well-behaved. A better approach, we claim, is to use *structural recursion* as the general technique for carrying sets into other structures. As opposed to general recursion, which most of the times requires destructors like *choose*, this form of programming works by matching arguments against data type constructors. One form of structural recursion on sets is given by the combinator Φ which takes $E : \beta$, $F : \alpha \to \beta$ and $U : \beta \times \beta \to \beta$ to the unique $\Phi(E, F, U) : \{\alpha\} \to \beta$ satisfying

$$\Phi(E, F, U)(\emptyset) = E$$
$$\Phi(E, F, U(\{x\}) = F(x)$$
$$\Phi(E, F, U)(s_1 \cup s_2) = U(\Phi(E, F, U)(s_1), \Phi(E, F, U)(s_2))$$

provided that on the range of $\Phi(E, F, U)$, U is associative and E is an identity for U (a monoid structure), and moreover that U is commutative and idempotent. This is similar to the *pump* operator of FAD [BBKV88] and the *hom* operator of Machiavelli [OBBT89], except that in those languages the requirement of idempotence is dropped and the requirement that the sets s_1, s_2 be disjoint in the third clause is added. *Pump* and *hom* have a natural denotational semantics, but their operational semantics is contrived. The evaluator must evaluate sets eagerly and then do time consuming dynamic tests for equality of values. Of course, this rules out working with sets of functions for example. Even for sets of, say, integers, mapping a function over a disjoint union may yield a non-disjoint one, which fed into *hom* would yield a run-time error. One would like to obtain statically an assurance that the program goes through, but it seems that only a few very simple programs can be shown correct in this sense. On the other hand *pump* and *hom* can be implemented in Φ style, by converting sets to *bags* and then doing structural recursion on those.

Appropriate uses of Φ are, for example, $\Phi_\cup(F) = \Phi(\emptyset, F, \cup)$ and $\Phi_\wedge(P) = \Phi(\text{true}, P, \wedge)$ where $F : \alpha \to \{\gamma\}$ and $P : \alpha \to \text{bool}$. Using these, we can construct the following functions on sets:

$$\text{map } f = \Phi_\cup(\lambda x.\{fx\})$$
$$\text{pairwith } s\ x = \text{map } (\lambda y.(x, y))\ s$$
$$\text{cartprod}(s_1, s_2) = \Phi_\cup(\text{pairwith } s_2)(s_1)$$
$$\text{powerset } = \Phi(\{\emptyset\}\ ,\ \lambda x.\{\emptyset\} \cup \{\{x\}\}\ ,\ \lambda(s_1, s_2).\ \text{map } \cup\ \text{cartprod}(s_1, s_2))$$

(checking the commutative-idempotent monoid requirement for the use of Φ in the last definition is quite interesting).

The denotational and operational semantics as well as an appropriate logic for reasoning about programs that compute with structural recursion over bulk data types such as lists, bags and sets is studied in [BS91], where transformations to other presentations of these datatypes are also given. In [BBN91], it is shown how to compute transitive closure efficiently with structural recursion, and it is noted that relational algebra can be characterized using restricted forms of structural recursion: the expressions of relational algebra are semantically equivalent to precisely those expressions that can be constructed using the structural recursors $\Phi_\cup$ and $\Phi_\wedge$ together with elementary operations (concatenation, projection and conditionals) on tuples.

4 Heterogeneous collections

The ability to deal with heterogeneous collections is claimed [Str87] as an important feature of object-oriented programming, and we believe it is of special importance in object-oriented databases, where it appears to be the only way to reconcile two natural views of inheritance [BO90]. Before looking at this issue we should remark that we have so far been working in a framework of *typed* languages. These are languages in which the only meaningful expressions are those that have a (declared or inferred) type. In such a language `3 + "cat"` is not a program because it has no type. Compare this with the situation in "dynamically typed" languages in which such expressions can be evaluated, but may yield run-time type errors. In any persistent programming language [AB87] it is desirable, for safety, to maintain a structure that describes the type of a database along with the database. However, in order to reason about these external types in a typed language requires some extra apparatus.

The need for this is seen in any language that has some form of subtype rule in conjunction with a bulk data type such as lists. If l is an expression of type *list*(*Person*) and e is an expression of

type *Employee*, the expression cons(e, l) that "inserts" e into l also has, because of the subtype rule, *list*(*Person*). The expression head(cons(e, l)) now has type *Person*, and we have "lost" some of the structure of e; more generally we can no longer use the equation e = head(cons(e, l)) to reason about our programs. In [BO90] we have proposed an extension of the *dynamic* types [ACPP89] in which values in a programming language are "views" that express the partial type of some completely typed object.

The way we achieve this is to incorporate into our type system a distinction between the *type* of an object and its *kind*. In object-oriented terminology the former specifies the class of an object and hence its exact structure, while the latter specifies that certain methods are available. In order to incorporate assertions on kinds of objects in the type system, we introduce a new form of assertion $e : \mathcal{P}(\kappa)$ denoting that e has the kind κ. For example, e:$\mathcal{P}$(<Name:string, Age:num>) means that at least `Name` and `Age` fields are available on e. <Name:string, Age:num> describes a *kind*, which we can think of as a set of types – the set of record types that contain `Name:string` and `Age:num` components.

Kinds are most useful in conjunction with heterogeneous collections, which may not have a uniform type, but may have a useful kind. For example, e:{$\mathcal{P}$(<Name:string, Age:num>)} means that e is a set of records, each of which has at least a `Name` and `Age` field, and therefore queries involving only selection of these fields are legitimate. To construct such a heterogeneous collections of uniform kind, an operation **filter** κ (S) is defined which selects all the elements of S which have the fields specified by κ and makes those fields available, i.e. **filter** κ (S):{$\mathcal{P}$(k)}.

An advantage of this approach is that it reconciles the database "isa" hierarchies (with extent inclusion) with the hierarchies of object-oriented languages (with method sharing.) To show this, let us assume that the following names have been given for kinds:

```
PersKind   for  <Name:string, Address:string>
EmpKind    for  <Name:string, Address:string, Sal:num>
```

Also suppose that `DB` is a set of type {$\mathcal{P}$(any)}. The meaning of `any` is the set of all possible types, so that we initially have no information about the structure of members of this set.

Since kinds denote sets of types, they can be ordered by set inclusion. In particular, `Empkind` is a "sub-kind" of `PersKind`. ¿From this, the inclusion **filter** `EmpKind` $(S) \subseteq$ **filter** `PersKind` (S) will always hold for any heterogeneous set S. This means that the "data model" (inclusion) inheritance is *derived* as a static property from an ordering on kinds rather than being something that must be achieved by the explicit association of extents with classes with dynamic maintenance of extents. Moreover, object-oriented (method sharing) inheritance is also derived from a polymorphic type of a method. For example, the type inference method we have described in section 2 guarantees that any polymorphic function applicable to $\mathcal{P}$(`PersKind`) is also applicable to $\mathcal{P}$(`EmpKind`). Thus, we achieve the desired coupling of the two forms of *is-a* in a static type system.

5 Conclusions

We have attempted to show that typed languages are a natural medium for many aspects of database programming languages. There are certain topics such as object identity, abstract types, views (and the interaction between these) that require further investigation. However we are confident that these can be resolved and that object-oriented databases will be best understood in the same framework of typed languages.

References

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

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), 1983.

[ABD+89] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrick, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. First Deductive and Object-Oriented Database Conference*, 1989.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *Transaction on Database Systems*, 10:230–260, 1985.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. ACM Symp. on Principles of Programming Languages*, 1989.

[BBKV88] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[BBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. Unpublished Manuscript, University of Pennsylvania, 1991.

[BO90] P. Buneman and A. Ohori. A type system that reconcile classes and extents. Technical report, University of Pennsylvania, 1990.

[BS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. To appear in *Proc. International Conference on Automata, Languages and Programming (ICALP)*, 1991.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

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

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. ACM Symp. on Principles of Programming Languages*, 1991.

[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, 1988.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. ACM SIGMOD Conference*, pages 424–434, 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[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, 1988.

[OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proc. ACM OOPSLA Conference*, pages 445–456, 1989.

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

[Oho90a] A. Ohori. Extending polymorphism to records and variants. Technical Report, University of Glasgow, 1990.

[Oho90b] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76:53–91, 1990.

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

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *Transactions on Database Systems*, 5(2), 1977.

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

[Str87] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. Symposium on Logic in Computer Science*, pages 37–44, 1987.