Inheritance and Persistence in Database Programming Languages

Peter Buneman
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA

Malcolm Atkinson
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

Abstract

In order to represent inheritance, several recent designs for database programming languages have made use of *class* construct, which can be thought of as a restricted data type with an associated set of instances Moreover, these classes are persistent they survive from one program invocation to another This paper examines whether it is necessary to tie together type, extent and persistence in order to model inheritance and suggests that they may be separated to provide more general database programming languages In particular we shall see that it is possible to assign a generic data type to a function that extracts all the objects of a given type in the database so that the class hierarchy can be derived from the type hierarchy We shall also examine object-level inheritance and its relationship to data types for relational databases A final section examines how the various forms of persistence interact with inheritance at both object and type level

Introduction

The concept of inheritance is now well established in databases [Smit77, Hamm81, Ship81], artificial intelligence [Brac79] and programming languages [Gold80], and it is therefore not surprising that several recent designs for database programming languages have incorporated some form of inheritance into their type system At the same time, several "knowledge base" extensions to existing languages have been developed to deal with inheritance, though usually in a way that is less directly integrated with the data types of the language In a recent survey of database programming languages [Atki85a], the authors - as a result of their own desire to construct a database programming language - attempted to survey past and current implementations and designs in order to understand what open problems remain One of the most interesting aspects of recent designs is how the various languages represent inheritance and how inheritance interacts with other components of the language The aim of this paper is to summarize some of the research problems in this area and to indicate that it may be possible to combine relational and object-oriented database programming within a uniform type system.

From the outset, even the terminology of inheritance is confusing When we try to relate concepts such as *Person* and *Employee*, we may use terms such as "subtype", "subclass", "specialization", "is-a" to imply that there is some ordering on these concepts However, there are at least three things that we may mean by this ordering

a) that any operation we can perform on a value of type *Person* can also be performed on a value of type *Employee*,

b) that the set of all *Employee* values in the database is constrained to be a subset of the set of all *Person* values in the database, or

c) that any *Employee* value can be created by adding information to some *Person* value

Whether or not these definitions are all really distinct depends, of course, on a precise definition of the terms "value", "type", "information" etc However, we

This work was partly supported by grants from the National Science Foundation (CER MCS82-19196) and the British Science and Engineering Research Council (GRC 86280, GRA 86541)

shall see that various languages do distinguish between these notions, moreover, it is an interesting and open question as to whether a programming language should attempt to tie them together

Another notion that is going to figure in this discussion is *persistence* Data base programming - and presumably knowledge base programming, if there is a difference - is distinguished by the requirement that data persist beyond the duration of a program In traditional programming languages, the only persistent structures are files, and the management of these structures is left largely to the operating system Some interactive programming languages have a form of all-or-nothing persistence by which an interactive session may be halted and resumed later, but between files and all-or-nothing persistence only a rather small number of languages have attempted implement persistence for more complex data structures In fact, one way of distinguishing database programming languages from other languages is by the mechanisms that they use to support persistence

In creating, say, an employee database in Pascal, our first step would surely be to create an *Employee* data type This alone will not suffice because Pascal has no direct means of keeping track of the *Employee* records we have created during the execution of a program We therefore create some further data structure, perhaps a linked list to maintain an *extent* for the type *Employee* which describes the set of all *Employee* records that are currently in the database Finally, we will want to ensure the persistence of the database by mapping it to a suitably persistent data type, such as a file Although the last two steps may often be combined simply by using files directly, this is not always the case and we should distinguish between a type, its extent, and its persistence Now in [Atki85a] the authors have argued that an ideal language would separate these three In fact they have claimed that persistence should be more properly associated with values rather than types - something that we shall review in the last section

To justify the claim that extent should be divorced from type, it is first of all obvious that there are many types, such as *Integer* for which a unique extent is almost useless We might well want to create a set of integers, but this set would certainly not contain all the integers that were created during execution of the program; moreover, we would very likely want to have several such sets. Even when we deal with types such as *Employee*, there are often cases for having multiple extents One may want to experiment with hypothetical states of the database, or, as a more practical example that we shall discuss later, one may want to create a new, temporary extent in order to improve the efficiency of a program by *memoizing* - storing intermediate results to avoid repeated computation

The last example is also an example of an extent which is not required to persist, and there are many more such examples especially in relational database programming, where one creates an intermediate, transient relation in order to simplify or optimize some larger computation Other examples of the use of non-persistent extents arise from using database-like data structures to do non database-like computations Merrett [Merr84] gives several examples of the use of relational algebra to solve a variety of problems drawn from areas as diverse as computational geometry and text processing

The first database programming languages made a clear separation between type, extent, and persistence In Pascal/R [Schmi77] one would construct an employee database by first declaring an *Employee* record type A declaration of the form

type *EmpRel* = **relation** **of** *Employee*,

then defines a **relation** type whose values provide extents The persistence of a relation is obtained by placing it in a database

var *Emp_DB* = **database**
Employees *EmpRel*

end,

where the type **database** behaves like a record type, but has persistence controlled in the same way that it is for *files* In Pascal/R there is a restriction that only *relation* data types can be placed in a database PS-algol [Atki83] takes a more general approach to persistence and allows arbitrary values to be placed in a database, it is also a straightforward matter to construct a generic set type in PS-algol to define extents

More recent database programming languages have not always maintained a clear separation of type, extent, and persistence, and the reason is certainly that this is more difficult to organize in the presence of inheritance Turning back to our possible characterizations of inheritance, in a normal database programming environment we would normally want the hierarchy on types to define the inclusion hierarchy on extents But if we do this, we are implicitly assuming that each type has a unique associated extent Does this mean that we have to forgo our desire to separate type from extent, or can

we find a more general framework that allows us the best of both worlds? The answer to this, which is not completely resolved, is discussed in the next section

Inheritance, Data Types and Extents

A number of attempts have been made to design database programming languages that exploit some form of inheritance, in particular there are object-oriented languages [Cope84] that implement persistent objects In the belief that, for databases, type-checking is one of the best techniques for ensuring program correctness, our main concern will be with languages whose type system is designed for predominantly *static* type-checking in the tradition of Pascal [Wirt81] However, in this context the types of Pascal are inadequate not only because they do not represent inheritance, but also because of the more general criticism that they cannot represent *generic* code, code that can be applied to values of more than one type. The second limitation has been taken care of in languages such as Ada [Ichb79] and ML [Gord79] which form the basis for several of the languages we shall discuss

The languages of interest to us also represent inheritance in some way In Taxis [Mylo80] for example, inheritance is fundamental, and programming constructs such as type, transaction, procedure, exception, set and record all have analogs in Taxis as classes, which are derived through some form of inheritance from a universal class Taxis, in fact, supports two forms of relationship among classes *instance* and *subclass*. For example the declaration

VARIABLE_CLASS EMPLOYEE **isa** PERSON **with**
characteristics
Emp_no: *integer*,
attribute_properties
Department: *char_8*,
.
end,

makes *EMPLOYEE* an instance of the meta-class *VARIABLE_CLASS*, whose instances have the property that they have an associated extent defined by explicit insertion and deletion It also makes *EMPLOYEE* a subclass of *PERSON* thereby ensuring that every instance of *EMPLOYEE* also has the attributes of an instance of *PERSON* If *PERSON* had also been defined as an instance of *VARIABLE_CLASS*, the declaration above would ensure that every instance of *EMPLOYEE* will be in the extent of *PERSON*.

In the other database programming languages, only the subclass (or subtype) hierarchical relationship was supported with any generality The distinction between the two hierarchies is nevertheless important. Some of the semantic network models used in Artifical Intelligence, e g KL-One [Brac85], distinguish between "is-a" and "is-a-kind-of" relationships, and in others they are confused For much of this paper we shall think of the instance (is-a-kind-of) hierarchy as having just two levels: type and value, or object and class However it is worth a brief digression to note that in database design we can move up and down the instance hierarchy quite easily This is illustrated by the following two scenarios, which are both based upon actual design problems

> The only information maintained on cars in the University parking lot is the registration number (tag), and make-and-model Information such as the length, which is used to derive charges and the availability of space, is derived from the make-and-model

Depending on the database management system which was to be used, one would probably employ a separate relation to hold information associated with make-and-model, or one would make make-and-model a compound attribute of car, but both of these solutions obscure the fact that a given car is an *instance* of a make-and-model type, and fail to represent properly the instance hiererchy In the second, and more mind-boggling scenario, the level in the instance hierarchy depends upon an attribute

> Products in a certain manufacturing plant that are above a certain price are treated as individuals and have attributes such as weight and completion date of construction. Below that price they are treated as classes and have weight and number in stock as properties of the class

Some formal work [Hull83] has suggested how these two hierarchies might be manipulated, but in database programming languages only Taxis appears to deal with the instance hierarchy and then only in a limited three-level framework In general, if we are to think of the value-type relationship in programming languages as an example of the instance hierarchy, and if we want to treat this as more than a two-level hierarchy, we will have to introduce the notion of meta-types, meta-meta-types, , something that is well beyond our practical understanding of types Let us therefore turn to the subtype or subclass hierarchy and use the convention that by *class* we mean a type with an associated

extent, noting that this is not what is meant by a *CLASS* in Taxis

There are several other (statically) typed database programming languages that support classes In Taxis, as we have seen, a *VARIABLE_CLASS* defines both a type and an extent. There is also an *AGGREGATE_CLASS* that is similar to *VARIABLE_CLASS*, but does not have an associated extent. One can think of *AGGREGATE_CLASS* as being similar to a record type in other programming languages Adaplex [Smit81] ties the notions of type and class together in a single **entity** type In Galileo [Alba85], one defines first a type and then uses the type to construct a class This is less restrictive, but it does not appear to be possible to construct two extents on the same type What is most interesting about Galileo is that the type upon which a class is based is not restricted, one may, for example, construct a class of integers In Adaplex and Taxis, the types associated with a class are restricted to being something like record types (although they are limited in the types that can be assigned to their components)

Designers of programming languages tend to be parsimonious in the number of constructs they introduce Among other things, a programming language with few constructs is probably simpler to implement and programs in that language are probably easier to reason about We therefore ask whether the notion of class is fundamental or whether it can be derived from more primitive constructs In particular we want to ask whether *class* is just a parameterized data type like **array** in Pascal That is, should we be able to write *class of* α in the same way that we can write **array of** α for any type α? To do this we shall contrast the three languages we have just mentioned with the programming language Amber [Card84] which supports inheritance on types and a very general form of persistence but which has no built-in class construct. Amber is in fact partly based on Galileo which in turn was derived partly from ML [Gord79]

To illustrate the distinction between Amber and the other languages, compare the Adaplex declaration for a person-employee database,

```
type Person is entity
    Name  String(1 32),
    Address

end entity,
 type Employee is entity
    Emp_no  Integer,
    Department  String(1 8)
end entity
 include Employee in  Person
```

with the corresponding declaration in Amber,

```
type Person is
    ≪Name  String, Address      ,    ≫
 type  Employee is Person with
    ≪Emp_no  Int, Dept  String,   ≫
```

Although syntactically similar, these declarations perform very different functions In the first place, type declarations in Amber such as those for *Person* and *Employee* serve only to create names for types Thus the Amber declaration for *Employee* above is equivalent to the declaration

```
type  Employee is
    ≪Name  String, Address      , . ,
     Emp_no  Int, Dept  String,  .≫
```

and it would still be inferred, from the structure of the definition, that *Employee* is a subtype of *Person* In Adaplex, types with the same structure are not necessarily identical, and the subtype hierarchy has to be explicitly defined by means of **include** directives

In Adaplex, the inclusion relationships among the extents associated with entity types follow directly from the explicit hierarchy of entity types Thus creating an instance of *Employee* will also create a new instance of *Person* In contrast, Amber, has no associated extents and it is not immediately obvious how to create a data structure that will provide the desired extents One way would be to represent the database as a list of values and then write functions that would extract the values of a given type. This requires us to maintain a list of heterogeneously typed values (the database) and to interrogate their types in order, say, to extract all the values of type *Employee*. Although Amber is a strongly typed language, the type-checking mechanism is not entirely static, there is a special type *Dynamic* whose values carry around both a value and a type Ordinary values, such as integers can be made dynamic by a **dynamic** operator, and coerced back to ordinary values with **coerce** Thus in

```
let d = dynamic 3,

let i = coerce d to Int,
let s = coerce d to String,
```

d is made a dynamic type *d* is not an integer, and any attempt to use an integer operation such as addition on *d* is a (static) type error The value in *d* can be "re-

vealed" by using **coerce** so that i, which is statically determined to be an integer, is bound to the value 3, but the subsequent line will raise a run-time exception because the type associated with d is not *string* Dynamic types, as we shall see, give us one method of treating persistence uniformly

We can therefore construct a database by creating a list of dynamic values, but we still need to be able to enquire about the types of these dynamic values in order, say, to extract all the *Employee* values in the database To do this, Amber provides a special type *Type* whose values describe types, and a special function *typeOf* that takes any dynamic value and returns a description (another value) of its type

Using these, it is possible to write functions of the form (this is not Amber syntax)

function *getPersons*(*d Database*)· *PersonList*,
function *getEmployees*(*d Database*)
EmployeeList,

where *PersonList*, *EmployeeList* and *Database* have been previously defined as lists containing values of respective type *Person*, *Employee* and *Dynamic* These functions therefore provide us with extents in the sense that *getPersons* will always return a larger list than *getEmployees*, and those records obtained by "projecting" the *Employee* records returned by *getEmployees* will always appear in the result of *getEmployees*

There are some drawbacks to this solution In the first place, this "database" is completely unconstrained we can put any dynamic value in it Secondly, this is not a very efficient solution since we have to traverse the whole database in order to obtain a small subset, we also have the overhead of having to check the structure of each value we encounter Another possibility would be to keep a set of (statically) typed lists with appropriate structure sharing, and further possibilities are discussed in [Chan82] This may solve some of the problems of efficiency, but it requires more elaborate functions and control mechanisms for creating new values and inserting them in the database But the main difficulty with either of these methods is that we have to write both the code for each *get* function and declare the type of each result list for all types in the database Moreover the code to decompose the type of a dynamic value can be lengthy Thus without the ability to write generic code, this approach cannot be considered practicable

What is required is a single generic *Get* function that would work for any type

function *Get*[*t*](*d Database*).*List*[*t*],

in which t is a type parameter that should be supplied in addition to the database parameter Thus we would write *Get*[*Employee*] to get a list of employees in the database of type *List*[*Employee*] Is there a type system powerful enough to allow us to write such generic code? In a recent paper, Cardelli and Wegner [Card85a] investigate the consequences of combining inheritance with various forms of type parameterization Although they do not deal directly with the problems of persistence or dynamic types, their type system does allow us to express the type of generic functions such as *Get*.

Consider an object-oriented language in which the result of *Get*[*Employee*](*d*) is a list of objects in the database Each object in this list necessarily has type *Employee*, but it may also have a type that is a subtype of *Employee*, e g. the object may also be of type *Student* How do we express the type of this object when we do not know exactly what it is? In the Cardelli-Wegner system we say that such an object o has type $\exists t\ t \leq Employee$ This reads "there exists a subtype t of employee such that o has type t" Note that what we are doing is specifying an *abstract* type we don't know what the type or representation of o is, all we know is that we can perform on o any operation associated with the type *Employee*

Just as we can use the metaphor of existential quantification to describe abstract data types, we can also use universal quantification to describe *polymorphism* Many languages, Pascal for example, have built-in polymorphic types such as *arrays* This means that for any type α the type **array. of** α is a valid type, and operations of access and update are defined for each type α Only recently have languages that allow user-defined polymorphic types been implemented These include ML, Poly [Matt85], Argus [Lisk83] and Ada [Ichb79] although in Ada one obtains the effect of a polymorphic type by use of a (generic) *package* In the Cardelli-Wegner system, the function *Cons*, which puts an element on the front of a list is of type $\forall\alpha\ (\alpha \times List[\alpha]) \rightarrow List[\alpha]$, which reads "*Cons* is a function that, for any type α, takes a value of type α and a value of type $List[\alpha]$, and returns a $List[\alpha]$". Note that our function *Get* is similarly polymorphic It is defined for any type t In fact, using both universal and existential quantification, we can write down the type of *Get* as

$\forall t\ Database \rightarrow List[\exists t'\ t' \leq t]$

What we have done is to show that with a sufficiently powerful type system, it is possible to write down the type of a function that extracts the objects of a given type from the database Moreover the use of this function can be type-checked *statically*, even though a certain amount of dynamic type-checking may be needed in the implementation Thus there is no need for a distinguished family of types for which inheritance is defined, nor is it necessary to have unique extents associated with these types

In order to provide a type system powerful enough to assign a type to functions like *Get*, a certain amount of computation has to take place at the level of types The compiler must be able to manipulate type expressions and decide if they are equivalent. Now the Cardelli-Wegner type system, while providing this power has the property that equality of type expressions is decideable, and there are no non-terminating computations at the level of types This is an obviously desirable property of type systems, but whether we can maintain this position for database programming languages is an open question

The problem arises with respect to data models Our traditional approach to database programming has been to combine an existing programming language with an existing data model Since the types of the language never match precisely the forms of data abstraction assumed in the data model, special-purpose modifications need to be made to the types of the language. Now we might ask if there is a sufficiently general notion of "type" in which we could directly express an arbitrary data model For example, we might ask for a type system in which we could write down the Entity-Relationship model [Chen76] or even a (simplified) Network data model as generic types Database schemata described by these models are represented as some form of labelled graph If we are to represent these as types, we require a type system that is powerful enough both to allow the representation of labelled graphs (as types, not values) and to allow the checking of integrity constraints such as acyclic conditions Such type systems have not yet been properly developed, and it is an open question whether they will be One solution is to treat types as values, this does not mean that one has to abandon type-checking, but does allow the possibility of non-terminating computations on types The question of whether types should be treated as values, thereby sacrificing some of the theoretical properties of decideable type systems is the subject of some recent debate [Burs84, Meye86] in programming language research

Inheritance on Values

Suppose we create an object o of type *Person* in the database and at some later time wish to extend this object so that it becomes an *Employee* object, o' There is a sense in which we have added information to o to create a more informative object o', and we can write $o \sqsubseteq o'$ to express the fact that o' contains more information than o Note that we have chosen an ordering on objects ($\sqsubseteq$) that is the reverse of the ordering ($\leq$) that we expect on their types a more informative object appears to have a type that is lower in the type hierarchy The database programming languages Adaplex, Galileo, Taxis that have built-in definitions of classes all have some method of extending an object so that it belongs to a new sub-class On the other hand in Amber two record values are *never* comparable, and there is no method of extending a record to become a more informative record The only way to transform a *Person* record into an *Employee* record would be to delete the less informative record and add a new one, and this may not be an equivalent operation when there are references to or from that record Again, special-purpose code would be required for each such transformation

Given two objects o and o' with $o \sqsubseteq o'$, there is an interesting question as to whether both o and o' should be allowed to exist simultaneously in the database According to the tenets of object-oriented programming [Cope84, Borg85], objects are not identified by intrinsic properties, so there is no reason why we should not allow two comparable objects to co-exist For example, suppose that, in the University parking lot example cited in the previous section, one did not keep registration (tag) information on cars One could then have two identical cars in the database, and to prevent the lot overfilling, one would want to maintain separate records for each car The problem of object identity is well-known in philosophy and is related to the distinction between types, or classes, and values that we have been discussing throughout this paper In ordinary discourse we can very easily switch levels in the instance hierarchy, e g "My car is a Chevvy Nova The Chevvy Nova weighs 3,000 pounds "so presumably we should be able to switch with same ease in databases If the type/value relationship is part of the instance hierarchy, we do not yet have the same flexibility of changing levels in programming languages

In database programming languages that are based

on the relational model, collections of objects are *sets* and it is therefore assumed that two objects (tuples) in a relation can only be distinguished by some intrinsic property. If we want to maintain the natural identity of tuples we usually impose natural or artificial *key* attributes on suitably chosen classes. Moreover the imposition of keys will also prevent comparable values (under $\sqsubseteq$) from coexisting in the same set. If, for example, we insist that *Name* is a key for *Person*, we cannot now place two comparable objects whose type is a subtype of *Person* in the database, for if they were comparable, they would necessarily have the same key.

The problem of object identity is one of the serious incompatibilities between object-oriented database programming and relational database programming. There are at least three important differences:

a) As we have seen, a relation is a *set* of tuples that are identified by intrinsic properties. One cannot give a tuple in a relation an independent identity.

b) There is no representation of inheritance in the relational data model.

c) Relations are *flat*. We cannot store complex structures such as arrays or other relations as values a relation. This is the well-known first-normal-form condition on relational databases.

In fact these differences had appeared irreconcilable. The first suggestion that it was appropriate to consider hierarchies in a relational setting was made by Zaniolo [Zani84a] in a treatment of null values in a relation. More recently Ait-Kaci, gave a precise characterization of inheritance and showed [AitK84] how one could actually use inheritance to provide a model of computation. (Although Ait-Kaci refers to this as type subsumption, there is in his formalism no distinction between objects and types, and the operations on types are equally applicable to values.) Zaniolo [Zani84b] has also suggested that "non-flat" data models may be more appropriate to express the relationship between databases and logic programming, and Bançilhon [Banç86] has shown how some of the relational database concepts can be used in connection with "non flat" data models.

Whether it is possible effectively to combine object-oriented with relational database programming is is not yet clear, but if we violate the flatness constraint of relational databases (c above) and violate the principles of object-oriented programming by not allowing comparable objects o and o' to exist simultaneously in a set (a above), we can make some headway. Here is a brief informal sketch of how this may be done. We can think of our objects as records such as

$$o_1 = \{Name = J\ Doe,\ Address = \{City = 'Austin'\}\}$$

in which the components may themselves be records. We may create a better defined record either by adding new fields or by better defining one of the existing fields, thus

$$o_2 = \{Name = 'J\ Doe',\ Address = \{City = 'Austin'\},\ Emp\_no = 1234\}$$

and

$$o_3 = \{Name = 'J\ Doe',\ Address = \{City = 'Austin', Zip = 78759\}\}$$

are both better defined than o_1. This is what we mean by "adding information", i.e. $o_1 \sqsubseteq o_2$ and $o_1 \sqsubseteq o_3$.

We mentioned above that one often wanted to add information to a record, e.g. one might want to turn a *Person* into an *Employee*. To do this, there must be a join operator $\sqcup$ that effectively merges the information in two records, thus

$$\{Name = 'J\ Doe'\} \sqcup \{Emp\_no = 1234\} = \{Name = 'J\ Doe', Emp\_no = 1234\}$$

As a more complicated example, the join of o_2 and o_3, $o_2 \sqcup o_3$, above is

$$\{Name = 'J\ Doe';\ Address = \{City = 'Austin', Zip = 78759\},\ Emp\_no = 1234\}$$

however, we cannot always join two records together since they may disagree on a common field. For example, we cannot join o_1 with $\{Name = 'K\ Smith'\}$, since there is no value we can put in the *Name* field that is "better than" both *'J Doe'* and *'K Smith'*.

From the foregoing we have seen informally that objects form a *partial order* under $\sqsubseteq$ and that there is a join operation $\sqcup$. More generally, it can be shown that this ordering is a *complete* partial order. In fact, this class can be extended to contain more general structures than records. For precise definitions of this ordering and further examples, see [AitK84] or [Banç86], but the reader is warned that these authors use lattices rather than complete partial orders.

The next step we take is to consider sets of such objects, and in keeping with our earlier discussion, consider only sets of mutually incomparable objects. We shall call a set of objects R a (generalized) *relation* if whenever $o_1, o_2 \in R$ then neither $o_1 \sqsubseteq o_2$ nor $o_2 \sqsubseteq o_1$ hold (sets with this property are called *cochains* in the

$$\{\{Name = \text{'J Doe'},\ Dept = \text{'Sales'},\ Addr = \{City = \text{'Moose'}\}\},$$
$$\{Name = \text{'M Dee'},\ Dept = \text{'Manuf'}\},$$
$$\{Name = \text{'N Bug'},\ Addr = \{State = MT\}\}\}$$

R_1

$$\{\{Dept = \text{'Sales'},\ Addr = \{State = WY\}\},$$
$$\{Dept = \text{'Admin'},\ Addr = \{City = \text{'Billings'}\}\},$$
$$\{Dept = \text{'Manuf'},\ Addr = \{State = MT\}\}\}$$

R_2

$$\{\{Name = \text{'J Doe'},\ Dept = \text{'Sales'},\ Addr = \{City = \text{'Moose'},\ State = WY\}\},$$
$$\{Name = \text{'M Dee'},\ Dept = \text{'Manuf'},\ Addr = \{State = MT\}\},$$
$$\{Name = \text{'N Bug'},\ Dept = \text{'Manuf'},\ Addr = \{State = MT\}\},$$
$$\{Name = \text{'N Bug'},\ Dept = \text{'Admin'},\ Addr = \{City = \text{'Billings'},\ State = MT\}\}\}$$

$R_1 \bowtie R_2$

Figure 1 A join of generalized relations

jargon of lattice theory) What this means informally is that we will not admit an object o into a relation R if there is already an object in R which contains as much information as o, and if it is more informative than objects already in R, we will subsume those objects in R We may now ask how relations themselves may ordered One way of defining an ordering on relations is to say that

> $R \sqsubseteq R'$ iff for every object o' in R' there is an object o in R such that $o \sqsubseteq o'$

In other words, $R \sqsubseteq R'$ if every object in R' is more informative than some object in R Again, $\sqsubseteq$ is a partial order on relations and we may ask if there is a corresponding join operation There is, and it is a generalization of the "natural join" for 1NF relations Figure 1 shows an example of such a join The same notation { } has been used for both sets and records This is because both structures can be derived from a more general structure, a *partial function*, and the orderings defined both on sets and on records are naturally derived from from the ordering on partial functions Moreover from a slightly different ordering on relations a a *projection* operator can be defined, and the interaction of these two orderings allows us derive the basic results of the theory of functional dependencies [Bune86] More interesting in this context is that the analysis makes no formal distinction between a type and a relation; in fact the type $\{Name : String; Age\ \ Int\}$ can be seen as a very large relation,

$$\{\{Name = s, Age = i\} \mid s \in String, Age \in Int\},$$

moreover it is meaningful to talk about the *join* of this relation with a relation R to extract all the objects in R whose type is a sub-type of $\{Name\ \ String; Age\ \ Int\}$ This is precisely the the operation of extracting sub-classes that we were attempting to define in the previous section

Whether what has been presented here is a forced marriage of relational and object oriented programming languages, or is a natural extension of both remains to be seen There are certainly a number of open problems. We have not given an account of keys for generalized relations, nor have we properly related object-level inheritance to inheritance on types described in the previous section. A partial solution to the latter problem is presented in [Bune85] where it is shown that a rather minor modification can be made to the type system of Amber to allow for object-level inheritance and to use this to assign a type to relational operators such as *join* Given the apparent connection both with relational database theory and with data types, this approach may bear fruit for database programming languages

Persistence and Extents

In [Atki85a] the authors advocated a more general view of persistence in which any value should be capable of persisting They argued that in most database systems only objects of certain types were allowed to persist, and this inhibited the use of database management in many applications such as the many branches of engineering in which special data structures have been developed for various forms of design (e g CAD/CAM) The first practical demonstration of a language in which any value could be made persistent was PS-algol [Atki83] Very few other database languages

have adopted this principle, of those we have discussed, only Galileo and Amber provide a uniform approach

In order to use persistence in a strongly typed environment, the authors suggested two principles that should govern the design of a database programming language

(1) Persistence is a property of values and should be independent of type

(2) While a value persists, so should its description (type)

The second condition guards against the possibility of writing out a data structure as one type and reading it in as another, a common cause of error in manipulating files in conventional programming languages

If we are to adopt this view that persistence is independent of type, there are at least three approaches we may take to defining persistence in programming languages The first, and simplest, is *all-or-nothing* persistence that is commonly used with interactive programming languages Some versions of Lisp and Prolog, for example, allow one to save the state of an interactive session and resume it later on This is usually achieved by copying a complete core image (or possibly an image of all user-defined structures) to secondary storage While simple to implement, this approach does not provide adequate structure for database work it does not allow sharing of values among programs, moreover the user cannot separate the relatively constant structures he has created (the database) from the extremely volatile structures such as experimental programs In this form of persistence the survival of the database is highly dependent on the integrity of the programming system as a whole

The second form of persistence is controlled by having program instructions that move structures in and out of secondary (persistent) storage We shall call this *replicating* persistence since structures are replicated in secondary storage In a sense, languages like Pascal offer some form of replicating persistence in file types However, as we have already seen, the structures that can be placed in files are limited, e g they cannot contain pointers, and they do not carry their own types Another example of replicating persistence is to be found in APL [Falk73] Here arbitrary values may persist and their type is carried with them, however APL's data structures are all *flat*, and their types relatively simple (functions are stored as character arrays) Amber provides the most complete example of replicating persistence through the use of dynamic types For example, the code in Amber to perform these operations is, approximately,

```
type database =
        -- Type declaration for a database
var d  database =
        -- Code to initialize the database
extern('DBFile', dynamic d),
```

and to access the database in a subsequent program,

```
type database =
        -- Type declaration for a database (as before)
var x = intern 'DBFile',
var d = coerce x to database
        -- Code to query or update the database
```

where the coerce operation would fail if the type associated with the dynamic value *d* were not *database* In Amber, when a dynamic value is *extern*ed, it carries with it everything that is reachable from that value such as structures that are referenced by that value or, in the case of function values, variables that are global to that function.

The name *DBFile* in this Amber example serves to maintain a name for a value across program boundaries, such names are called *handles* However, in this case the handle refers to a copy of the data in the program To see what this means, in the program

```
var x = intern 'DBFile',
        -- Code that modifies x
x = intern 'DBFile',
```

the modifications to *x* will not survive the second intern operation. Thus if any concurrency is to be implemented through the use of replicating persistence, it must be done by ensuring that the various *extern* and *intern* operations for a given handle are properly synchronized Also, under this form of persistence, if values *a* and *b* both refer to a third value *c* then any change made to *c* though a handle for *a* will not be visible from a handle for *b*, since these two handles will refer to distinct copies of *c* This may be the cause of both update anomalies and wasted storage

The third form of persistence is what we shall call *intrinsic* persistence Here the idea is that every value in a program is persistent, however there is no need physically to retain storage for values for which all reference is lost In this model of persistence there is no need to replicate data or control its movement, nor is there any distinction in the programming language between primary and secondary storage The physical represen-

tation of a value is determined entirely by the run-time support for the programming language it could be that all values are maintained in secondary storage even during program execution, although this would not be an efficient way of implementing intrinsic persistence

The entire purpose of handles for this form of persistence is to maintain reference to values To give an example of how this might work, consider the following hypothetical program in "Persistent Pascal"

program *Test*,
type *DBType* = ,
var *DB DBType* **handle** *DBHandle*,
begin

The sole purpose of *DBHandle* is to provide a name for the value *DB* that is global to the program *Test* But for the fact that we might want to use different internal names in different programs, there would be no harm in simply marking *DB* as the global name. Creating this global name is all that is required to ensure persistence, there is no need for any *extern* or *intern* operations during the execution of the program

What we have just given is an idealized description of intrinsic persistence PS-algol [Atki83] and Gemstone [Cope84] implement some form of intrinsic persistence, however our description here ignores some important points In the first place we have implicitly assumed a single global namespace Although it is global to the program, is it also global to the user, the user community, ? In practice one needs to operate with multiple namespaces and control the sharing of structures among namespaces As another practical matter we need to protect ourselves against a program failing when the database is in an undesirable state, therefore PS-algol provides an explicit *commit* instruction Before this instruction is called, the persistent value and the value being used by the program can diverge

In spite of the inadequacies of this description of persistence, let us look at some of issues in type-checking Assuming static type-checking, the first time the program *Test* is compiled, the type *DBType* is associated with the handle *DBHandle* Now suppose that at a later time, we recompile a modified version of *Test* with a new definition *DBType'* for the type of *DB* There is no reason why the compilation will fail if *DBType* is a subtype of *DBType'* In fact, by the definition of subtype, the program should work since all the operations defined for *DBType'* must be applicable to the value associated with the handle *DBHandle* This second compilation with *DBType'* is simply providing us with a view of the data

A more interesting possibility arises when *DBType* is not a subtype of *DBType'*, but is *consistent* with it, i e there is a common subtype of both *DBType* and *DBType'* As a result of the second compilation, the handle now refers to a value with a richer structure. Provided we never contradict any of our previous definitions, we can can continue to enrich the type, or schema, of the database We should note that intrinsic persistence is appropriate to this form of typechecking, since the obvious interpretation of an *extern* operation for an object of type *DBType'* is to replicate an object of that type rather than a supertype, thereby losing structure from the database

While it is desirable to have as much static type-checking as possible, Atkinson & Morrison [Atki85b] argue that some dynamic type-checking in database programming languages is necessary, and a propose a mechanism that provides both Another important issue is whether persistence should be a property of what we normally consider *modules* as opposed to values In the Cardelli-Wegner type system this is not a problem since one of the main contributions of this work is to demonstrate that the combination of inheritance and existential types allows us to treat modules as values However there are certain penalties that one pays for this In particular the type associated with a module is necessarily abstract, one cannot get at its implementation Moreover there are certain forms of module parameterization that appear to be desirable in database programming but that cannot be represented in this type system. Some of these issues are discussed in [Card85], but the general problem of what form of module or type parameterization is appropriate for database programming, and how this interacts with persistence is still an open question

A final and rather simple observation concerns the relationship of object-level inheritance and persistence It is that adding transient information to a persistent structure can be quite useful One of the examples used in [Atki85a] is a bill-of-materials computation This is a text-book excercise but proved rather awkward in some of the languages that were examined It is required simultaneously to compute the cost of manufacturing and total mass of a manufactured part. We shall examine the simpler problem of computing just the total cost of manufacturing a part A standard recursive program to do this is, in outline,

function *TotalCost(p Part)*,
  if *p.IsBase* **then** *p PurchasePrice*
  else *p ManufacturingCost* +

*sum{TotalCost(q SubPart)*q Qty|*
q **on** *p Components}*,

where the *Components* of a part *p* is a list of records that describing the part and quantity for each subpart used in the manufacture of *p* The only difficulty with this is that when a given subpart is used in more than one way in the manufacture of a larger part, the total cost will be needlessly recomputed for that subpart. This will happen when the parts explosion diagram is not a tree but a directed acyclic graph.

The way out of this is to memoize intermediate results. In order to do this we need to attach further fields to the *Part* type in which to store these results, we also need to modify the function *TotalCost(p)* so that it first checks these fields to see if it has already done the computation for the part *p*. Now these additional fields are not required to be accessible outside the the computation of *TotalCost*. Even though the *Part* values in which we are interested are presumably persistent, there is no need for the additional information to persist.

Conclusions

When the authors first undertook a survey of database programming languages, they had a general impression that most of the problems in producing a uniform persistent language with a type system appropriate to databases were related to implementation rather than design. It now appears that many of the issues brought out by language design, especially in the areas of inheritance, types and persistence, are central to programming language research as a whole The purpose of this paper has been to convey some of the open research areas in the hope of stimulating the database community to think about them

Acknowledgements

The authors are greatly indebted to David Maier for his careful and constructive criticisms of a draft of this paper Alex Borgida and Ron Morrison also made numerous helpful comments

References

[AitK84] Ait-Kaci, H. "A Lattice Theoretic Approach to Computation based on a Calculus of Partially Ordered Type Structures", PhD. Dissertation, Department of Computer and Information Science, Moore School/D2, University of Pennsylvania, Philadelphia, PA 19104 (1984)

[Alba85] Albano, A , Cardelli, L and Orsini, R , "Galileo A Strongly Typed Interactive Conceptual Language", ACM Transactions on Database Systems, 10, 2, March 1985.

[Atki83] Atkinson, M.P , Bailey, P.J , Chisholm, K J , Cockshott, W.P. and Morrison, R , "An Approach to Persistent Programming", Computer Journal, 26, 4, November 1983

[Atki85a] Atkinson, M.P. and Buneman, O P. "Database Programming Language Design", Technical Report 10-85, University of Pennsylvania

[Atki85b] Atkinson, M.P and Morrison, R., "Types, Bindings and Parameters in a Persistent Environment", Proceedings of the Appin Conference on Data Types and Persistence, Technical Report, Department of Computing, Glasgow University, September 1985.

[Banç86] Bançilhon, F and Khoshafian, S , "A Calculus for Complex Objects", ACM Conference on Principles of Database Systems, May 1986

[Brac79] Brachman, R.J "On the Epistemological Status of Semantic Networks", *Associative Networks - The Representation of Knowledge in Computers*, N V Findler, ed., Academic Press, New York, 1979.

[Brac85] Brachman, R.J and Schmolze, J G , " An Overview of the KL-One Knowledge Representation System", Cognitive Science, 9,2, April 1985

[Burs84] Burstall, R and Lampson, B , "A kernel language for abstract data types and modules", Proceedings of the international symposium on semantics of data types, Sophia-Antipolis, France, June 1984.

[Borg85] Borgida, A ' "Features of Languages for the Development of Information Systems at the Conceptual Level", IEEE Software, 2, 1, January 1985.

[Bune85] Buneman, O.P., "Data Types for Data Base Programming", Proceedings of the Appin Conference on Data Types and Persistence, Technical Report, Department of Computing, Glasgow University, September 1985.

[Bune86] Buneman, O.P, "A Domain Theoretic Approach to Relational Databases", Technical Report, University of Pennsylvania Department of Computer and Information Science, January 1986

[Card84] Cardelli, L., "Amber", AT&T Bell Labs Technical Report, 1984.

[Card85a] Cardelli, L and Wegner, P, "On Understanding Types, Data Abstraction, and Polymorphism", Technical Report, Brown University, Aug 1985

[Card85b] Cardelli, L and MacQueen, D.M, "Persistence and Type Abstraction", Proceedings of the Appin Conference on Data Types and Persistence, Technical Report, Department of Computing, Glasgow University, September 1985.

[Chan82] Chan,A, *et al.*, "Storage and Access Structures to Support a Semantic Data Model" Proceedings VLDB, Mexico City, 1982.

[Chen76] Chen,P P S, "The Entity-Relationship Model Towards a Unified View of Data", ACM Transactions on Database Systems, 11, 1, March 1976

[Cope84] Copeland, G. and Maier, D, "Making Smalltalk a Database System", Proceedings ACM Sigmod, Boston, June 1984

[Falk73] Falkoff, A.D and Iverson, K.E, "The design of APL", IBM Journal of Research and Development, 10, July 1973.

[Gold80] Goldstein, I P. and Bobrow, D. G., "Extending object oriented programming in Smalltalk", Proceedings of the 1980 Lisp Conference, August, 1980.

[Gord79] Gordon, M.J, Milner, A.J.R.G, and Wadsworth, C P, *Edinburgh LCF*, Springer-Verlag, Lecture Notes in Computer Science, 1979

[Hamm81] Hammer, M. and McLeod, D, "Database Description with SDM A Semantic Database Model", ACM Transactions on Database Systems, 6, 3, Sept 1981

[Hull83] Hull, R. and Yap, C.K, "The Format Model A theory of Database organisation", 1st ACM symposium on Principles of Database Systems, 1982.

[Ichb79] Ichbiah *et al*, "Rationale of the Design of the Programming Language Ada", ACM Sigplan Notices, 14, 6, 1979

[Lisk83] Liskov, B., Herlihy, M, Johnson, P, Leavens, G., Scheifler, R and Weihl W, "Preliminary ARGUS reference manual", MIT LCS Memo 39, October 1983

[Matt85] Matthews, C.J, "Poly Manual", University of Cambridge, Computer Laboratory, Technical Report 63, February 1985

[Merr84] Merrett, T H ,*Relational Information Systems*, Reston Publishing Co., 1984

[Meye86] Meyer, A R, and Reinhold, M B, "'Type' is not a Type· Preliminary Report", Proceedings 1986 ACM Conf on Principles of Programming Languages, February 1986

[Mylo80] Mylopoulos, J., Bernstein, P A and Wong, H.K.T, "A Language Facility for Designing Database Intensive Applications", ACM Transactions on Database Systems, 5, 2, June 1980

[Schm77] Schmidt, J W, "Some High Level Language Constructs for Data of Type Relation", ACM Transactions on Database Systems, 2, 3, September 1977.

[Ship81] Shipman, D W., "The Functional Data Model and the Data Language DAPLEX", ACM Transactions on Database Systems, 6, 1, March 1981.

[Smit77] Smith, J.M. and Smith, D C.P, "Database Abstractions - Aggregation and Generalization" ACM Transactions on Database Systems, 2, 2, June 1977

[Smit81] Smith,J M, Fox,S and Landers,T., "Reference Manual for ADAPLEX", Computer Corporation of America, January 1981

[Wirt81] Wirth, N, "The Programming Language PASCAL", Acta Informatica, 1, 1971

[Zani84a] Zaniolo, C, "Database Relations with Null Values", JCSS, 28 1, pp 142-166, February 1984

[Zani84b] Zaniolo, C "Prolog A Database Query Language for All Seasons", Proc IEEE-ACM International Expert Database Systems Workshop, Kiawah Island, October 1984