SEPTEMBER 1989 VOL. 12 NO. 3

a quarterly bulletin of the
Computer Society of the IEEE
technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON DATABASE PROGRAMMING LANGUAGES

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE

IEEE COMPUTER SOCIETY

The LOTUS Corporation has made a generous donation to partially offset the cost of printing and distributing four issues of the Data Engineering bulletin.

Message from the TC Chair

It gives me great pleasure to announce that Professor Sham Navathe of the University of Florida has accepted to serve as our TC's representative to the Standards Activities Board of the IEEE Computer Society. We look to Dr. Navathe for leadership in helping our TC to participate in standards activties related to Data Engineering. If you are interested in working with Sham, please contact him at:

Professor Sham Navathe
Computer and Information Sciences
Weil Hall
University of Florida
Gainesville, FL 32611
Phone: (904) 335-8456
E-Mail: sham@bikini.cis.ufl.edu

By this time all of you should have received a letter from me requesting your Dues Payment for membership in the TC on Data Engineering. The IEEE CS Board of Governors approved our request to charge dues. We need your support on this matter to be able to bring you our Bulletin and to continue to sponsor and co-sponsor the many conferences that push the state-of-the-art in Data Engineering.

The Board of Governors has determined that in the future all TCs will have to be run in a more business-like manner based on dues and the profits from conferences and other sponsored events. So our Dues Policy is actually an experiment for all TCs to watch. Please take the time to make out your check to help our efforts in building a financially strong TC.

If you haven't paid yet, you will be receiving another letter from me in September. For our non-USA colleagues, we now have a credit card payment system in place. It will be explained in the next mailing.

Finally I have been receiving E-Mail and letters from many of you who want to get more involved in TC activities. As opportunities arise I will be contacting you.

Best Regards,
Larry Kerschberg

Letter from the Editor

"Database Programming Languages" is the theme of this issue of **Data Engineering**. The database programming language (DBPL) area is a broad and very active area of research; work in the area addresses problems faced by those who must design, implement, and maintain large, data-intensive applications. The seven papers included in this issue are extended abstracts of full papers from the *Second International Workshop on Database Programming Languages*, which was held on the Oregon coast this past June. This set of seven papers was recommended by the workshop's program committee as a good set to give the **Data Engineering** readership a taste of the broad range of problems and solutions that were addressed at the workshop. For readers who would like to learn more about the DBPL area, the full proceedings of the workshop will be available shortly as:

Proceedings of the Second International Workshop on Database Programming Languages, Hull, Stemple, and Morrison, eds., Morgan-Kaufmann Publishers, Inc., San Mateo, CA (1989).

Since the co-chairs of the DBPL program committee kindly contributed a nice overview of both the area and the workshop, I will turn things over to them at this point. However, before I do, I would like to thank each of the authors for agreeing to write extended abstracts for this special issue of **Data Engineering**. I would also like to thank the three DBPL program committee co-chairs, particularly Rick Hull, for helping to make the issue possible. Enjoy!

Michael J. Carey
August, 1989

Current Directions in Database Programming Languages

Work in database programming languages represents an approach to integrating the technologies and paradigms of programming languages and database management in order to address the problems of modern data-intensive applications. The applications that are driving these efforts tend to have combinations of the following attributes:

- large amounts of complex, shared, concurrently accessed, persistent data
- reliability requirements
- distribution of data storage and processing over networks
- design orientation, e.g., computer-aided design of complex artifacts such as circuits, manufactured goods and software
- complex behavior often involving inference or rule-based computation
- sophisticated graphical interfaces.

A number of successful approaches, each with its subculture of practitioners, have been advanced for relatively narrow combinations of these concerns: Database Management Systems deal realiably with large amounts of distributed persistent data; Computer-Aided Design uses special standalone software packages (e.g., software engineering environments and VLSI design systems) written in standard programming languages with file systems; Expert System Shells support certain kinds of inferencing; and Object-Oriented Systems facilitate the development of systems with complex data or sophisticated graphical interfaces.

A database programming language (DBPL) is meant to be a facility addressing all the aspects of performing sophisticated computations over large amounts of complexly structured, distributed, shared, reliable, persistent data. One of the first forums for studying the problems of integrating these technologies was provided by the First International Workshop on Persistence and Data Types, held in Appin, Scotland in August 1985, and organized by Malcolm Atkinson, Peter Buneman and Ron Morrison. This workshop spawned four others, including the First and Second International Workshops on Database Programming Languages; another two are scheduled for 1990 and 1991.

At present there is no consensus that all aspects of the DBPL problem can or should be addressed in one language. The research presented at the 2nd DBPL workshop focused largely on particular aspects of the problem, for example developing the felicitous addition of one paradigm's features to those of another, or exploring ways to extend the capabilities of modelling tools such as type systems and storage managers. The abstracts included in this issue of Data Engineering provide a small but representative sample of the papers presented at the workshop, and indicate some of the leading research efforts currently being pursued in the DBPL area.

A major issue in the workshop concerns type systems for DBPLs. Two of the abstracts included here indicate the current breadth of the study in this area. The abstract by Ohori, Buneman and Breazu-Tannen proposes Machiavelli, a polymorphic language with static type inference which encompasses several of the features of object-oriented database systems. (Actually, the more complete version of this paper appears in SIGMOD 89 rather than the workshop; a companion piece in the workshop provides a philosophic discussion of the difficulties of extending static type checking to other object-oriented and DBPL features.) The abstract by Jacobs presents a type system for logic programming langauges, and can be used as the basis for typing algebraic database programming languages. Another group of papers included in the workshop attempts to provide a framework for comparing database programming language type systems, and then characterizes four such systems.

Another major topic of the workshop was persistence in relation to other programming langauge and database capabilities. Included here is an abstract by Agrawal and Gehani, which indicates some of the

issues involved in adding persistence to a programming language, and presents the solutions proposed in the language O++, an extension of C++. Another paper of the workshop offers an alternative approach for adding persistence, in that case for ADA; and another illustrates how object-orientation can be built on top of a persistent language. A paper proposing a novel implementation strategy for handling large quantities of persistent objects was also presented.

Several papers of the workshop addressed the mismatch between set-oriented and item-at-a-time modes of database languages and programming languages (respectively). Three papers of the workshop propose query languages for object-oriented database systems, the abstract of Shaw and Zdonik included here indicates one of the approaches taken. All three use "complex objects" (closely related to nested relations), but differ on issues of the creation of new types and the breaking of object-oriented encapsulation. Another workshop paper addressed the issue of extending compiler technology to the set-oriented operators of DBPLs. Various other combinations of object-orientation, deductive databases, and functional programming in the context of DBPLs were also explored at the workshop.

The issue of generalizing logic databases received significant attention at the workshop. The abstract by Chen, Kifer and Warren included here presents an approach for extending the syntax of a deductive language to higher order while leaving the semantics essentially first order. A second proposal in the same vein was included in the workshop, and also a proposal which supports a higher-order semantics. The abstract by Naqvi here argues that "stratification" can serve as a design principle for dealing with a variety of extensions to basic Horn clause logic.

Another issue addressed in the workshop is that of providing tools for layered database system development. The abstract by Borgida et. al. included here stems from the DAIDA project, whose goal is to build an environment for constructing data-intensive software. As described here, the system will use three languages: an "assertional knowledge representation languages", an intermediate design language, and an underlying DBPL. A second paper of the workshop describes the PROQUEL language, an integrated specification, data manipulation and programming langauge also intended for use in a layered development enviroment.

A final group of papers of the workshop, not represented here, attempts to expand and refine our understanding of DBPL issues by providing taxonomies and/or new perspectives. In addition to the papers mentioned above which provide a framework for comparing type systems, this includes papers which: raise global issues such as type evolution arising in the context of persistent types; categorize different kinds of inheritance; contrast object identity vs. reference; compare the use of objects vs. structured values; and which formally analyze the expressive power and complexity of various DBPL primitives.

A great deal of research is still needed in the field of DBPLs. In addition to the topics mentioned above, active research is being pursued in the areas of so-called "long transactions" and collaborative working, query optimization, and integrity constraint maintenance. In the coming years experimentation with the various paradigms for DBPLs will continue; it is difficult to predict what approaches will ultimately predominate.

Richard Hull
Ron Morrison
David Stemple

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 some object-oriented database systems, type errors are not caught until something goes wrong at run-time, often with disastrous consequences. To the best of our knowledge, none of the systems or research prototypes developed in the past exploits the flexibility offered by type inference.

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 between a program object and a real-world object was maintained by endowing the program object with

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

"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 the experimental language Machiavelli [OBB89] that has been implemented at the University of Pennsylvania. Moreover, this paper summarizes some of the issues discussed in more detail in [BBO89].

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 [JM88] for a related study). 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. We want its type to be something like $\{\tau_1\} \times \{\tau 2\} \rightarrow \{\tau_1 \sqcup \tau_2\}$, such that when τ_1 and τ_2 are record types which

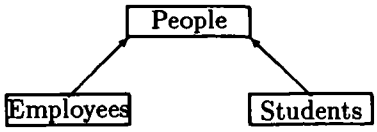

Figure 1: A Simple Class Structure

have *consistent* type information in their common fields, $\tau_1 \sqcup \tau_2$ is the *union* of the field descriptions of τ_1 and τ_2. (Throughout the paper, $\{\tau\}$ will stand for the type of *sets* of elements of type τ.) Note that the 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**. (In ML, ! stands for *dereferencing.*) 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 has a salary (is an employee):

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

(ML has a convenient "undefined" value, whose type is **unit**.) This is not a type declaration in Machiavelli

(type declarations are in general not needed). *PersonObj* 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 $\mathtt{project}(View_\sigma(S), \tau) \subseteq View_\tau(S))$ where $View_\tau$ and $View_\sigma$ denote the corresponding viewing functions on classes τ and σ, and where $\leq$ is the order relation generated by field description inclusion. 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. Besides the usual union of sets of the same type, the primitive operation **unionc** takes an argument of type $\{\delta_1\} \times \{\delta_2\}$ for all non-functional types δ_1, δ_2 such that $\delta_1 \sqcap \delta_2$ exists (the types are compatible on the common fields). Let s_1, s_2 be two sets having types $\{\delta_1\}, \{\delta_2\}$ respectively. Then $\mathtt{unionc}(s_1, s_2)$ satisfies the following equation:

$$\mathtt{unionc}(s_1, s_2) = \\ \mathtt{project}(s_1, \delta_1 \sqcap \delta_2) \cup \mathtt{project}(s_2, \delta_1 \sqcap \delta_2)$$

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, **unionc(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 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 some *ad hoc* encoding is not entirely satisfactory. 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

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

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

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

(Here, as before, $\leq$ is the relation generated by field description inclusion, which is the *dual* of the inheritance relation of [CW85].) 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 $\overline{\tau_1} = \{\tau | \tau \leq \tau_1\}$ and $\overline{\tau_2} = \{\tau | \tau \leq \tau_2\}$. 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 $\overline{\tau_1} \cap \overline{\tau_2} = \overline{\tau_1 \sqcup \tau_2}$, 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 trivial. For example, the function that extracts the *Name* field from a record is implemented by the following polymorphic term:

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

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. 2nd 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.

[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.

A Type System for Algebraic Database Programming Languages *

Extended Abstract

Dean Jacobs †
University of Southern California

July 11, 1989

1 Introduction

There is a growing body of research which uses algebraic theory as a framework for integrating notions from databases and programming languages. The fundamental idea in this work is that database states are represented by first-order declarative programs. Functions and predicates within such programs are used uniformly to define the underlying data types, represent values stored in the database, and compute derived data. Database updates are performed by rewriting the program which represents the state. This work might be generally classified as the study of *algebraic database programming languages.*

This extended abstract describes a type system for logic programs which is intended to serve as the basis for algebraic database programming languages; a more complete description appears in [Jac89]. This type system supports user–defined types, parametric polymorphism, union types, and subtypes. As an example, the declarations

```
type etree :- {nil}.
type netree(A) :- {node(tree(A),A,tree(A))}.
type tree(A) :- etree + netree(A).
```

introduce the type `etree` of all empty trees, the type `netree(A)` of all non–empty trees, and the type `tree(A)` of all trees. A type represents a set of ground terms, the basic values of computation in logic programming, e.g., `etree` represents the singleton set {`nil`}. Polymorphic types, such as `tree(A)` have type parameters, e.g., `tree({red})` represents the set

$$\{\texttt{nil}, \texttt{node}(\texttt{nil}, \texttt{red}, \texttt{nil}), \texttt{node}(\texttt{node}(\texttt{nil}, \texttt{red}, \texttt{nil}), \texttt{red}, \texttt{nil}), \ldots\}$$

These declarations arrange it so that `etree` and `netree(A)` are subtypes of `tree(A)`. Subtypes are intended to capture the semantic database notion of isa; elements of `netree` have all the properties of elements of `tree`.

Our type system has a particularly interesting treatment of type equivalence and the subtype relationship. There are two basic notions of the conditions under which type τ_1 should be considered to be equivalent to type τ_2: structural equivalence, where τ_1 and τ_2 must have the same meaning, and name equivalence, where τ_1 and τ_2 must be textually identical. Similarly, there are two basic notions of the conditions under which τ_1 should be considered to be a subtype of τ_2: structural subtyping, where the meaning of τ_1 must be appropriately contained in the meaning of τ_2, and prescriptive subtyping, where τ_1 must be explicitly declared to be a subtype of τ_2. In our system, predefined type constructors, such

*This research was supported in part by AT&T grant 5345990979

†Comp. Sci. Dept., University of Southern California, Los Angeles, CA 90089-0782, jacobs@pollux.usc.edu

as +, are treated structurally and user defined type constructors, such as `tree`, are treated by name and prescriptively. For example, `tree({red})` is a subtype of `tree({red}+{blue})`, however, given the declaration `type elist :- {nil}.`, `elist` is not a subtype of `tree({red})`.

We present Milner–style type inference rules for logic programs which can serve as the basis for type–checking algorithms. These rules rely on our notions of type equivalence and subtypes and on the notion of a typing for the symbols in a program. The typing for function symbols is derived from type declarations and the typing for predicate symbols is explicitly specified by predicate declarations. We briefly discuss a type-checking algorithm which infers the typing of variables.

Type systems for logic programs may be generally classified as being either descriptive or prescriptive. In a descriptive type system, types and the typing of predicate and function symbols are automatically inferred [Mis84, MR85, Red88, Zob87, YS87]. The goal here is to derive safe upper bounds, so called "optimistic" types, for the purposes of compile–time optimization. In a prescriptive type system, types and the typing of predicate and function symbols are explicitly declared by the programmer. The goal here, as in most conventional type systems, is to provide documentation, to catch certain basic errors, and to support optimization. Prescriptive type systems may be further classified as being either semantic or syntactic. In a semantic type system, the underlying logic contains an explicit notion of types and the computational mechanisms are correspondingly enhanced [GM86, Smo88, HV87, Wal88]. Order–sorted logic and order–sorted unification have been extensively studied in this context. In a syntactic type system, the underlying logic is untyped and conventional computational mechanisms are used [MO84, DH88]. Here, types are introduced at compile–time simply for the purposes of type checking.

The type system presented in this paper is prescriptive and syntactic. We extend the type system of [MO84], which is an adaptation of Milner's polymorphic type system for ML [Mil78] to logic programming, to include union types and subtypes. [DH88] also extends [MO84] to include subtypes, however, their work is somewhat problematic. In particular, they use a form of structural subtyping and it is undecidable in their system whether one type is a subtype of another.

This paper is organized as follows. Section 2 reviews some basic terminology and definitions associated with logic programming. Section 3 defines the syntax and semantics of types and type declarations. Section 4 introduces type equivalence and subtypes. Section 5 introduces the notion of a typing for the symbols in a program. Section 6 presents type inference rules for logic programs. Section 7 presents concluding remarks.

2 Preliminaries

The language of first order predicate logic has the following countable sets of symbols.

- V of variables, ranged over by x
- F of function symbols with given arity, ranged over by f/n
- P of predicate symbols with given arity, ranged over by p/m

A term is either a variable or an n-ary function symbol applied to n terms. An atom is an n-ary predicate symbol applied to n terms. A clause h:-b consists of an atom h, called the head, and a list of atoms b, called the body. A logic program $C_1, \ldots, C_n; q$ is a collection of n clauses C_i, called the rule base, and a list of atoms q, called the query.

A substitution θ is a mapping from variables to terms. An explicitly enumerated substitution over variables $\{x_1, \ldots, x_j\}$ is written $[x_1 \mapsto t_1, \ldots, x_j \mapsto t_j]$. The application of substitution θ to term t, denoted $t\theta$, is the term obtained from t by replacing all occurrences of variables in the domain of θ by their associated terms. Term t_1 has substitution instance t_2, denoted t_1 **ins** t_2, iff there is a substitution θ such that $t_1\theta = t_2$. Terms t_1 and t_2 are equivalent up to renaming of variables, denoted t_1 **rnm** t_2, iff t_1 **ins** t_2 and t_2 **ins** t_1. If t_1 **rnm** t_2 then there is a renaming substitution θ, i.e., a substitution which maps variables to variables, such that $t_1\theta = t_2$. The set of variables occurring in term t is denoted $\mathbf{var}(t)$; $\mathbf{var}(t_1, t_2)$ abbreviates $\mathbf{var}(t_1) \cup \mathbf{var}(t_2)$.

Term t is said to be ground iff $\mathbf{var}(t) = \emptyset$. Ground terms, such as `node(nil,red,nil)`, are the values over which computation takes place in our system. We let $\mathcal{H}$ denote the set of all ground terms, i.e., the Herbrand Universe.

3 Types and Type Declarations

Let the following countable sets of symbols be given.

- TV of type variables, ranged over by α
- TC of type constructors with given arity, ranged over by c/k

A type $\tau \in Type$ is an expression inductively built up from variables $\alpha \in TV$, type constructors $c \in TC$, function symbols $f \in F$, and the union operator $+$. For example, `A`, `tree(A)`, `{red}`, `tree({red})`, `{red}+{blue}`, and `tree({red}+{blue})` are all types.

Intuitively, a type represents a set of ground terms, i.e., a member of $\mathcal{T} = \mathcal{P}(\mathcal{H})$. Formally, the semantics of a type is given in terms of an assignment to type variables and an interpretation for type constructors. A *type variable assignment* $\eta \in \mathcal{TVA}$ is a partial function from TV to $\mathcal{T}$. A *type constructor interpretation* $\zeta \in \mathcal{TCI}$ is a partial function over $c/k \in TC$ such that $\zeta(c) : \mathcal{T}^k \rightarrow \mathcal{T}$.

Definition 1 (Semantics of Types) *The function* $\mathcal{M} : Type \rightarrow \mathcal{TVA} \times \mathcal{TCI} \rightarrow \mathcal{T}$ *defines the semantics of types as follows.*

$$\begin{array}{l}
\mathcal{M}[\![\alpha]\!](\eta,\zeta) = \eta(\alpha) \\
\mathcal{M}[\![c(\tau_1,\ldots,\tau_k)]\!](\eta,\zeta) = \zeta(c)(\mathcal{M}[\![\tau_1]\!](\eta,\zeta),\ldots,\mathcal{M}[\![\tau_k]\!](\eta,\zeta)) \\
\mathcal{M}[\![\{f(\tau_1,\ldots,\tau_n)\}]\!](\eta,\zeta) = \{f(t_1,\ldots,t_n) \mid t_1 \in \mathcal{M}[\![\tau_1]\!](\eta,\zeta) \wedge \ldots \wedge t_n \in \mathcal{M}[\![\tau_n]\!](\eta,\zeta)\} \\
\mathcal{M}[\![\tau_1 + \tau_2]\!](\eta,\zeta) = \mathcal{M}[\![\tau_1]\!](\eta,\zeta) \cup \mathcal{M}[\![\tau_2]\!](\eta,\zeta)
\end{array}$$

The members of a collection of types are usually interpreted with respect to a fixed $\zeta \in \mathcal{TCI}$. We say ground term t *has type* τ, denoted $t : \tau$, iff $t \in \mathcal{M}[\![\tau]\!](\eta,\zeta)$ for all $\eta \in \mathcal{TVA}$ over $\mathbf{var}(\tau)$. For example, t : `tree(A)` iff t = `nil`; this suggests that free variables in a type may be viewed as being "universally quantified". We say type τ_1 is a *structural subtype* of τ_2, denoted $\tau_1 \preceq \tau_2$, iff $\mathcal{M}[\![\tau_1]\!](\eta,\zeta) \subseteq \mathcal{M}[\![\tau_2]\!](\eta,\zeta)$ for all $\eta \in \mathcal{TVA}$ over $\mathbf{var}(\tau_1,\tau_2)$. For example, it is not the case that `tree(A)` $\preceq$ `tree(B)`. We say type τ_1 is *structurally equivalent* to type τ_2, denoted $\tau_1 \simeq \tau_2$, iff $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_1$.

A type declaration for $c/k \in TC$ has the form

$$\mathbf{type}\ \ c(\alpha_1,\ldots,\alpha_k) : - \tau.$$

where the α_i are distinct type variables. We say a set of type declarations has domain $D \subseteq TC$ iff it contains a declaration for each member of D and it refers only to type constructors in D. A type constructor interpretation ζ over D is said to be a *model* of a set of type declarations with domain D iff for every type declaration as above we have $\tau \preceq c(\alpha_1,\ldots,\alpha_k)$. The meaning of a set of type declarations is taken to be the intersection of its models, i.e., its least model.

We now present some example type declarations. Our first example illustrates enumerated types.

```
type primary :- {red} + {blue} + {yellow}.
type secondary :- {purple} + {green} + {orange}.
type color :- primary + secondary.
```

Here, `red` : `color`, `purple` : `color`, and `primary` $\preceq$ `color`.

A marked polymorphic union type can be defined as follows.

```
type munion(A,B) :- {first(A)} + {second(B)}.
```

Here, `first(red)`:`munion(color,B)` and `munion(primary,B)` $\preceq$ `munion(color,B)`.

A polymorphic list type can be defined as follows.

```
type elist :- {nil}.
type nelist(A) :- {A.List(A)}.
type list(A) :- elist + nelist(A).
```

Here, the cons operation "." is written in infix form to improve readability. A particular member of this type has lists nested to a fixed depth, e.g.,
`red.nil:list(color)`
`(red.nil).(blue.nil).nil:list(list(color))`
A single type where lists can be nested to an arbitrary depth can be defined as follows.

```
type xlist(A) :- list(A) + xlist(xlist(A)).
```

Along these same lines, a type of lists of any type can be defined as follows.

```
type anylist :- list(A).
```

This definition ensures `list(A)` $\preceq$ `anylist`, i.e., $\mathcal{M}[\![\texttt{list(A)}]\!](\eta,\zeta) \subseteq \mathcal{M}[\![\texttt{anylist}]\!](\eta,\zeta)$ for all η over {A}.

4 Type Equivalence and Subtypes

The following definition introduces a binary relation $\equiv$ which determines whether two types are considered to be equivalent for the purposes of type checking. As described in section 1, predefined type constructors, such as +, are treated structurally and user defined type constructors, such as `list`, are treated by name.

Definition 2 (Type Equivalence) $\equiv$ *is the smallest congruence relation on types which satisfies the following properties.*

1. $\tau + \tau \equiv \tau$.
2. $\tau_1 + \tau_2 \equiv \tau_2 + \tau_1$.
3. $(\tau_1 + \tau_2) + \tau_3 \equiv \tau_1 + (\tau_2 + \tau_3)$.
4. $f(\ldots, \tau_1 + \tau_2, \ldots) \equiv f(\ldots, \tau_1, \ldots) + f(\ldots, \tau_2, \ldots)$.

Recall that a congruence relation is reflexive, symmetric, transitive, and allows substitution of equal terms, i.e., if $\tau_1 \equiv \tau_2$ then $\tau[\alpha \mapsto \tau_1] \equiv \tau[\alpha \mapsto \tau_2]$. We can show that $\equiv$ respects structural equivalence in the following sense. If $\tau_1 \equiv \tau_2$ then $\tau_1 \simeq \tau_2$ for all $\zeta \in \mathcal{TCI}$ over $\mathbf{var}(\tau_1, \tau_2)$.

We now introduce a binary relation $\mathbf{isa}^*$ which determines whether one type is considered to be a subtype of another for the purposes of type checking. As described in section 1, predefined type constructors, such as +, are treated structurally and user defined type constructors, such as `list`, are treated prescriptively. A set of type declarations defines a type hierarchy isa as follows: each type declaration

$$\mathbf{type}\ c(\alpha_1, \ldots, \alpha_k) :- \tau.$$

introduces τ $\mathbf{isa}$ $c(\alpha_1, \ldots, \alpha_k)$. The relation $\mathbf{isa}^*$ is defined to be the closure of isa as follows.

Definition 3 (Closure of a Type Hierarchy) *The closure* $\mathbf{isa}^*$ *of type hierarchy* isa *over domain* $D \subseteq TC$ *is the smallest relation over types from* D *which satisfies the following properties.*

1. *If* τ_1 $\mathbf{isa}$ τ_2 *then* $\tau_1\theta$ $\mathbf{isa}^*$ $\tau_2\theta$ *for any renaming substitution* θ.
2. *If* $\tau_1 \equiv \tau_2$ *then* τ_1 $\mathbf{isa}^*$ τ_2.
3. τ_1 $\mathbf{isa}^*$ $\tau_1 + \tau_2$
4. *If* τ_1 $\mathbf{isa}^*$ τ_2 *and* τ_2 $\mathbf{isa}^*$ τ_3 *then* τ_1 $\mathbf{isa}^*$ τ_3.

5. *If* τ_1 **isa*** τ_2 *and* τ_1' **isa*** τ_2' *then* $\tau_1[\alpha \mapsto \tau_1']$ **isa*** $\tau_2[\alpha \mapsto \tau_2']$.

We can show that **isa*** respects structural subtyping in the following sense. Let ζ be the type constructor interpretation associated with a set of type declarations and **isa** be the associated type hierarchy. If τ_1 **isa*** τ_2 then $\tau_1 \preceq \tau_2$.

5 Typings for Symbols

In this section, we introduce the notion of a *typing*, a binary relation "::" which associates each function symbol, predicate symbol, and variable in a program with "extended" types.

Function symbols are associated with extended types of the form $(\tau_1, \ldots, \tau_n \rightarrow \tau)$ where $\tau_1, \ldots, \tau_n, \tau$ are ordinary types. The function symbol typing associated with a program is derived from its type declarations as follows. Each disjunct $\{f(\tau_1, \ldots, \tau_n)\}$ appearing at the outermost level of a type declaration, i.e.,

$$\textbf{type } c(\alpha_1, \ldots, \alpha_k) : - \ldots + \{f(\tau_1, \ldots, \tau_n)\} + \ldots.$$

produces $f :: (\tau_1, \ldots, \tau_n \rightarrow \{f(\tau_1, \ldots, \tau_n)\})$. As an example, the type integer can be defined as follows.

```
type nat :- {0} + {succ(nat)}.
type unnat :- {0} + {pred(unnat)}.
type int :- nat + unnat.
```

These declarations produce the typings

$$\begin{aligned} &\texttt{0} :: (\rightarrow \{\texttt{0}\}) \\ &\texttt{succ} :: (\texttt{nat} \rightarrow \{\texttt{succ(nat)}\}) \\ &\texttt{pred} :: (\texttt{unnat} \rightarrow \{\texttt{pred(unnat)}\}) \end{aligned}$$

Function symbols may be overloaded, for example, the declaration

```
type gnat :- {0} + {succ(gnat)}.
```

produces

$$\texttt{succ} :: (\texttt{gnat} \rightarrow \{\texttt{succ(gnat)}\})$$

Predicate symbols are associated with extended types of the form $(\tau_1, \ldots, \tau_m)$ where $\tau_1, \ldots, \tau_m$ are ordinary types. The predicate symbol typing associated with a program is given by explicit declarations of the form

$$\textbf{pred } p(\tau_1, \ldots, \tau_m).$$

For example, the predicate *append* on lists might be declared as follows.

```
pred append(list(A), list(B), list(A+B)).
```

Predicate symbols may be overloaded.

Variables are associated with ordinary types and may not be overloaded. This restriction is necessary to ensure that each variable occurs in at most one type context. As an example of the problems which can arise if this restriction is not observed, consider the following predicates.

```
pred p(int).
     p(0).
pred q(color).
     ...
```

If we allowed `X :: int` and `X :: color`, then the query `:- p(X), q(X).` would be well-typed, however, it can lead to the type-incorrect resolvent `:- q(0).`

6 Type Inference Rules

In this section, we present type inference rules for logic programs. These rules allow us to derive expressions consisting of program components annotated with types. A program component is said to be *well-typed* if an annotation of it can be derived. A well-typed program is guaranteed not to produce type errors during its execution. We briefly discuss an algorithm, based on the inference rules, for determining whether a program is well-typed. In the following, the fact that expression e can be derived is denoted $\vdash e$.

As a first step, we present inference rules for terms t which allow us to derive expressions of the form $t : \tau$. The notation $\sigma_1 \lhd \sigma_2$, meaning function type σ_1 is *compatible* with function type σ_2, is defined as follows: $\sigma \lhd (\tau_1, \ldots, \tau_n \rightarrow \tau)$ iff there exists a substitution instance $(\tau_1', \ldots, \tau_n' \rightarrow \tau')$ of σ such that $\tau_i' \equiv \tau_i$ and τ' isa* τ.

Definition 4 (Inference Rules for Terms)
Terms: $f/n \in F$

$$\frac{f :: \sigma,\ \sigma \lhd (\tau_1, \ldots, \tau_n \rightarrow \tau),\ \vdash t_i : \tau_i}{\vdash f(t_1, \ldots, t_n) : \tau}$$

Variables: $x \in V$

$$\frac{x :: \tau_1,\ \tau_1 \ \mathbf{rnm}\ \tau_2}{x : \tau_2}$$

We can show that these inference rules are consistent with the semantics of types in the following sense. Suppose $t \in \mathcal{H}$ and $\vdash t : \tau$, then $t : \tau$.

We now present inference rules for atoms. These rules allow us to derive expressions of the form $p(t_1 : \tau_1, \ldots, t_m : \tau_m)$ where $p/m \in P$. There are two such rules; one for atoms in the head of a clause and one for atoms in the body of a clause.

Definition 5 (Inference Rules for Atoms)
Head Atoms: $p/m \in P$

$$\frac{p :: \sigma,\ \sigma\ \mathbf{rnm}\ (\tau_1', \ldots, \tau_m'),\ \tau_i' \equiv \tau_i,\ \vdash t_i : \tau_i}{\vdash p(t_1 : \tau_1, \ldots, t_m : \tau_m)}$$

Body Atoms: $p/m \in P$

$$\frac{p :: \sigma,\ \sigma\ \mathbf{ins}\ (\tau_1', \ldots, \tau_m'),\ \tau_i' \equiv \tau_i,\ \vdash t_i : \tau_i}{\vdash p(t_1 : \tau_1, \ldots, t_m : \tau_m)}$$

The difference between the rules for head and body atoms is that the parameters of a head atom may not be instantiated.

A logic program is well-typed if every atom in it is well-typed. The following proposition states that these inference rules are sound in the sense that every resolvent produced during execution of a well-typed program is well-typed, thus execution cannot "go wrong".

Proposition 1 (Soundness II) *Let G be a query $:\!-A_1, \ldots, A_i$, C be a clause $B :\!- B_1, \ldots, B_j$ such that A_1 and B have an mgu θ, and G' be a resolvent $:\!-B_1\theta, \ldots, B_m\theta, A_2\theta, \ldots, A_i\theta$ of G and C. If C and G are well-typed, then G' is well-typed.*

An (inefficient) algorithm for determining whether a program is well-typed can be derived from these rules by applying them in a top-down manner. This algorithm requires a typing for function and predicate symbols and derives a typing for variables. To process an atom $p(t_1, \ldots, t_m)$ where $p :: (\tau_1, \ldots, \tau_m)$, check if t_i has a type which is structurally equivalent to τ_i. Instantiation of variables in τ_i may occur if the atom appears in the body. If the atom appears in the head, variables in τ_i should be effectively viewed as being ground. To check if $f(t_1, \ldots, t_n)$ has type τ where $f :: \sigma$, find a type $(\tau_1, \ldots, \tau_n \rightarrow \tau)$ such that $\sigma \lhd (\tau_1, \ldots, \tau_n \rightarrow \tau)$ and then check if t_i has type τ_i. The set of all types carried down to instances of a variable must be unifiable and the type of the variable is taken to be the unification of this set. This algorithm requires a considerable amount of backtracking.

7 Concluding Remarks

The next step in this research is to provide support for the dynamic aspects of data modeling. This can be accomplished by introducing the notion of a class as a set of values of some underlying type. Each such value will play the role of an object identifier for an instance of its class. Attributes of objects can then be modeled as predicates on the underlying types of classes. For example, the declaration

```
class person of nat with
   name:string
   parent:person
end person
```

might be translated into

```
pred person(nat).
pred name(nat,string).
pred parent(nat,nat).
```

This allows the notion of inheritance to follow naturally from the notion of subtype.

The issue of "class checking", as distinct from type checking, now arises. As an example, suppose the user wishes to assert the tuple `parent(X,Y)` for some ground variables `X` and `Y`. Type checking ensures that the underlying type of the new values is correct. In addition, class checking must be performed to ensure that `person(X)` and `person(Y)` hold. This can be accomplished by analysing the source of the values for `X` and `Y`.

References

[DH88] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for prolog. In *Proc 2nd European Symposium on Programming, Springer LNCS 300*, 1988. Nancy, France.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, Englewood Cliffs, 1986.

[HV87] M. Huber and I. Varsek. Extended prolog for order-sorted resolution. In *4th IEEE Symposium on Logic Programming*, pages 34–45, 1987. San Francisco.

[Jac89] Dean Jacobs. A type system for algebraic database programming languages. In *Proc of the 2nd Workshop on Database Programming Languages*. Morgan Kaufmann, 1989.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computers and System Science*, 17(3):348–375, 1978.

[Mis84] P. Mishra. Towards a theory of types in prolog. In *Proc of the International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MR85] P. Mishra and U.S. Reddy. Declaration-free type checking. In *Proc Symposium on Principles of Programming Languages*, pages 7–21. ACM, 1985.

[Red88] Uday S. Reddy. Notions of polymorphism for predicate logic programs. In *Proc of the 5th International Symposium on Logic Programming*. IEEE, 1988.

[Smo88] Gert Smolka. Logic programming with polymorphically order-sorted types. In *Proceedings 1st International Workshop on Algebraic and Logic Programming*, 1988. Gaussig, GDR.

[Wal88] C. Walther. Many-sorted unification. *Journal of the ACM*, 35(1):1–17, 1988.

[YS87] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. In Ehud Shapiro, editor, *Concurrent Prolog Vol. 2.* MIT Press, 1987.

[Zob87] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proc of the 4th International Symposium on Logic Programming*, pages 817–838. IEEE, 1987. Melbourne.

Design of the Persistence and Query Processing Facilities in O++: The Rationale*

R. Agrawal
N. H. Gehani

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Ode is a database system and environment based on the object paradigm [2]. The database is defined, queried, and manipulated using the database programming language O++ which is an upward compatible extension of the object-oriented programming language C++ [31] that offers one integrated data model for both database and general purpose manipulation. The C++ object definition facility is called the *class*, which supports data encapsulation and multiple inheritance. O++ extends C++ by providing facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

In this paper, we present the O++ facilities for persistence and query processing, the alternatives that we considered, and the rationale behind our design choices. We assume that the reader is familiar with the basic issues in the design of database programming languages (see, for example, [7, 10, 11, 23, 29, 34] for an introduction) and concentrate on our design. Our discussion is oriented around O++, but we think that lessons we learned have wider applicability.

2. OBJECT DEFINITION: A BRIEF OVERVIEW

The C++ object facility is called the class. Class declarations consist of two parts: a specification (type) and a body. The class specification can have a private part holding information that can only be used by its implementer, and a public part which is the type's user interface. The *body* consists of the bodies of functions declared in the class specification but whose bodies were not given there. For example, here is a specification of the class `item`:

```
class item {
    double wt; /* in kg */
public:
    Name name;
    item(Name xname, double xwt);
    double weight_lbs();
    double weight_kg();
};
```

The private part of the specification consists of the declaration of the variable `wt`. The public part consists of one variable `name`, and the functions `item`, `weight_lbs` and `weight_kg`.

C++ supports inheritance, including multiple inheritance [32], which is used for object specialization. The specialized object types inherit the properties of the base object type, i.e., the data and functions, of the "base" object type. As an example, consider the following class `stockitem` that is derived from class `item`:

---

* This paper is an abbreviated version of the paper titled *Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++* presented at the *2nd International Workshop on Database Programming Languages* [3].

```
class stockitem: public item {
   int consumption;
   int leadtime;
public:
   int qty;
   double price;
   stockitem(Name iname, double iwt, int xqty, int xconsumption,
             double xprice, int xleadtime);
   int eoq();        /*economic order quantity*/
};
```

`stockitem` is the same as `item` except that it contains other information such as the quantity in stock, its consumption per year, its price and the lead time necessary to restock the item.

3. PERSISTENCE

We visualize memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent store and they continue to exist after the program creating them has terminated. A database is a collection of persistent objects, each identified by a *unique* identifier, called the object identity [16]. We shall also refer to this object identity as a *pointer to a persistent object.*

3.1 Design Goals

When incorporating persistence in O++, we kept the following design goals in perspective:

- Persistence should be orthogonal to type [7]. Persistence should be a property of object instances and not types. It should be possible to allocate objects of any type in either volatile or persistent store.
- There should be no run-time penalty for code that does not deal with persistent objects.
- Allocation and manipulation of persistent objects should be similar to the manipulation of volatile objects. For example, it should be possible to copy objects from persistent store to volatile store and vice versa in much the same way as it is possible to copy objects from the stack to the heap and vice versa.
- Inadvertent fabrication of object identities should be prevented.
- Language changes should be kept to a minimum.

3.2 Persistence in O++

3.2.1 Allocating and Manipulating Persistent Objects Persistent objects are referenced using pointers to persistent objects (that is, their identities). Persistent objects are allocated and deallocated in a manner similar to heap objects. We choose to view persistent storage as being similar to heap storage because most programmers are already familiar with manipulating heap objects. Persistent storage operators `pnew` and `pdelete` are used instead of the heap operators `new` and `delete`. Here is an example:

```
persistent stockitem *psip;
...
psip = pnew stockitem(initial-values);
```

`psip` is allocated on stack but `pnew` allocates the `stockitem` object in persistent store and its id (returned by `pnew`) is saved in `psip`. Note that `psip` is a pointer to a persistent `stockitem` object, and *not* a persistent pointer to a `stockitem` object.

Persistent objects can be copied to volatile objects and vice versa using simple assignments. Components of persistent objects are referenced like the components of volatile objects.

3.2.2 Dual Pointers Having only ordinary pointers and pointers to persistent objects has the following ramifications:

i. If a class is used to build a linked data structure, then the same class cannot be used to create the data structure both in volatile memory and in persistent store.
ii. We cannot write a function that can take as arguments pointers to either volatile objects or to persistent objects.

We avoid these problems by providing `dual` pointers that can reference either a volatile object or a persistent object. Whether the pointer refers to a volatile object or to a persistent object is determined at run time. Here is an example illustrating the use of dual pointers:

```
class node {
    ...
    dual node *next;
public:
    ...
    dual node *add(dual node *n);
};

persistent node *p, *proot; /*proot refers to a persistent list*/
node *v, *vroot;            /*vroot refers to an ordinary list */
...
proot = proot->add(p);
vroot = vroot->add(v);
```

3.3 Some Alternatives

O++ has three kinds of pointers: pointers to volatile objects (ordinary C or C++ pointers), pointers to persistent objects, and dual pointers. We were reluctant to add two new pointer types and gave much thought to the following two alternatives:

i. Have only one type of pointer that can point to both volatile and persistent objects.
ii. Have two types of pointers: an ordinary pointer that can only point to volatile objects and another that can point either to a persistent object or a volatile object, i.e., a dual pointer.

3.3.1 One-Pointer Alternative Using one pointer to refer to either a volatile object or a persistent object has several advantages: the language has two less pointer types and all (including existing) code will work with both volatile objects and persistent objects. This code compatibility could be at the source level or even at the object level depending upon the implementation strategy. Having one pointer type also makes types strictly orthogonal to persistence. Finally, the only syntactic additions required to C++ would be the operators `pnew` and `pdelete`.

We rejected the one-pointer alternative mainly on performance grounds: we wanted to have facilities that were easy to port and efficient to implement on general purpose machines and on standard operating systems (particularly on the UNIX® system) without requiring special hardware assists.

With the one-pointer alternative, a run-time check must be made to determine whether a pointer refers to a volatile object or a persistent object. Performing this check for every pointer dereference imposes a run-time penalty for references to volatile objects. This overhead is unacceptable in languages such as C and C++ (and therefore O++) which are used for writing efficient programs. Note that in these languages pointers are used heavily, e.g., string and array manipulation are all done using pointers.

One way of avoiding this run-time overhead is to trap references to persistent objects, for example, by using illegal values such as negative values for pointers to persistent objects (assuming that the underlying hardware or operating system provides the appropriate facilities). References to volatile objects then would not incur any penalty. But this scheme has several problems as discussed in [3].

3.3.2 Two-Pointer Alternative Instead of using three pointers, we seriously considered using only two pointers: ordinary pointers and dual pointers, but finally decided in favor of the three pointer approach. The main argument for not having pointers that point only to persistent objects is that the gain in run-time efficiency by avoiding the check needed in dual pointers would not be significant compared to the

cost of accessing persistent store. On the other hand, the use of separate pointers for persistent objects leads to better program readability and allows the compiler to provide better error checking, e.g., flagging arithmetic on pointers to persistent objects as being inappropriate. Based on our limited experience (which is writing small sample programs in O++), we feel that programmers will use pointers to persistent objects when appropriate.

3.4 Clusters of Persistent Objects

We view persistent store as being partitioned into clusters each of which contains objects of the same type. Initially, we decided that there would be a one-to-one correspondence between cluster names and the corresponding type names. Whenever a persistent object of a type was created, it was automatically put in the corresponding cluster. Thus, our clusters were type extents [12].

This strategy preserved the inheritance relationship between the different objects in the persistent store, and worked nicely with our iteration facilities (discussed in the next section) allowing us to iterate over a cluster, or over a cluster and clusters "derived" from it[1]. Before creating a persistent object, the corresponding cluster must have been created by invoking the `create` macro (in the same program or in a different program). Additional information (such as indexing) may be provided to the object manager to assist in implementing efficient accesses to objects in the cluster. A cluster, together with all the objects in it, can be destroyed by invoking the `destroy` macro.

Bloom and Zdonik [10] discuss issues in using type extents to partition the database. Although this scheme frees the programmer from explicitly specifying the cluster in which a persistent object should reside, it suffers from the following disadvantages:

i. Unrelated objects of the same type will be grouped together.
ii. It will not be possible to optimize queries that involve subsets of the objects in a cluster.

In short, the disadvantages were due to the absence of a subclustering mechanism.

Although we used type extents to partition the database, our scheme did not suffer from the above problems because subclusters could be created with the inheritance mechanism. For example, employees of a companies A and B could be represented by types `A-employee` and `B-employee` derived from the type `employee`. Each of these derived types would have a cluster corresponding to it. All employees could then be referenced using the cluster `employee`, while employees belonging only to company A or company B could be referenced by using clusters `A-employee` or `B-employee`, respectively. Different indices could also be created for the two clusters.

Reactions (from our colleagues) to this clustering scheme was unfavorable on the grounds that our clustering scheme did not allow multiple clusters for the same object type. The use of the inheritance mechanism for creating subclusters was not appealing for the following reasons:

i. The type inheritance mechanism is a static mechanism.
ii. The type inheritance mechanism is being overloaded to create subclusters.
iii. The use of type inheritance to form subclusters becomes unwieldy if the number of subclusters is large.

Consequently, we allowed creation of multiple clusters to group together objects of the same type, but retained the notion that some "distinguished" clusters represent type extents to preserve the inheritance relationship between objects in the database. An object implicitly always belongs to the distinguished cluster whose name matches the name of its type. However, it may also belong to another cluster which may be thought of as a subcluster of the corresponding distinguished cluster.

---

1. To be precise, clusters are not derived; they simply mirror the inheritance (derivation) relationship between the corresponding classes.

Subclusters are also created and destroyed with the `create` and `destroy` macros. A subcluster name is a string qualified by the corresponding cluster (object type) name, e.g.,

```
create(employee::"A-employee");
destroy(employee::name);          /*name is a string variable*/
```

Objects may be specified to belong to a specific subcluster when they are allocated in persistent store, e.g.,

```
ep = pnew employee("Jack")::"A-employee";
```

4. QUERY PROCESSING CONSTRUCTS

4.1 Design Goals

An important contribution of the relational query languages was the introduction of the set processing constructs. This capability allows users to express queries in a declarative form without concern for physical organization of data. A query optimizer (see, for example, [28]) is made responsible for translating queries into a form appropriate for execution that takes into account the available access structures. Object-oriented languages, on the other hand, typically do not provide set-oriented processing capabilities. Indeed, the major criticism of the current object-oriented database systems is that the query processing in these systems "smells" of pointer chasing and that they may take us back to the days of CODASYL database systems in which data is accessed by "using pointers to navigate through the database" [18]. One of the design goals of O++ was to provide set-processing constructs similar to those found in relational query languages.

Object-oriented languages are "reference" oriented, in that the relationship between two objects is established by embedding the identity of one object in another. Frequently, it is not possible to envision all relationships between the objects at the time of designing the database schema (class definitions). Relational systems are "value" oriented and relationships between objects (tuples) are established by comparing the values of some or all attributes of the objects involved. This lack of capability to express arbitrary "join" queries has been cited as another major deficiency of the current object-oriented database systems [18]. A design goal of O++ was to correct this deficiency.

In [7], Atkinson and Buneman proposed four database programming tasks as benchmarks to study the expressiveness of various database programming languages. One of the tasks, the computation of the bill of materials which involves recursive traversal, was found particularly awkward to express in many of the database programming languages discussed. Considerable research has been devoted recently to developing notations for expressing recursive queries in a relational framework and designing algorithms for evaluating them (see, for example, [1, 8]). Providing a capability to express recursive queries in a form that can be used to recognize and optimize recursive queries was another design goal of O++.

4.2 Set-Oriented Constructs

Recall that the persistent objects of a type implicitly belong to the corresponding distinguished cluster which has the same name as the type. Objects in a cluster can also be partitioned into named subclusters. O++ provides a `for` loop for accessing the values of the elements of a set, a cluster or a subcluster:

`for` *i* `in` *set-or-cluster-or-subcluster* [*suchthat-clause*] [*by-clause*] *statement*

The loop body, i.e., *statement*, is executed once for each element of the specified set, the cluster or the subcluster; the loop variable *i* is assigned the element values in turn. The type of *i* must be the same as the element type.

The *suchthat-clause* has the form `suchthat` (e_{st}) and the *by-clause* has the form `by` (e_{by} [, *cmp*]). If the `suchthat` and the `by` expressions are omitted, then the `for` loop iterates over all the elements of the specified grouping in some implementation-dependent order. The `suchthat` clause ensures that iteration is performed only for objects satisfying expression e_{st}. If the `by` clause is given, then the

iteration is performed in the order of non-decreasing values of the expression e_{by}. If the by clause has only one parameter, then e_{by} must be an arithmetic expression. An explicit ordering function *cmp* can also be supplied as the second argument; in this case, expression e_{by} need not be arithmetic.

For example, the following statement prints the name of stockitems heavier than 10kg in order of increasing price:

```
for s in stockitem suchthat(s->weight_kg() > 10) by (s->price)
        printf("%s %f\n", s->name, s->price);
```

We expect to pass the suchthat and by clauses to the object manager to select only the desired object ids and deliver them in the right order for the for loop.

To allow expression of order-independent join queries, we allow multiple loop variables in the for loops:

for i_1 in *set-or-cluster-or-subcluster*$_1$, ..., i_n in *set-or-cluster-or-subcluster*$_n$
[suchthat (e_{st})] [by (e_{by})] *statement*

Such loops allow the expression of operations with the functionality of the arbitrary relational join operation. For example, we can write

```
for e in employee, d in dept suchthat(e->dno==d->dno && e->salary>100)
    printf("%s %s\n", e->name, d->name);
```

to print the names of the employees who make more than 100K and the names of their departments.

When iterating over a set or a cluster, we allow iteration to also be performed over the elements that are added during the iteration, which allows the expression of recursive queries [4]. We hope to recognize recursive queries and use efficient algorithms for processing them.

4.3 Accessing Cluster Hierarchies

All objects in hierarchically related clusters can be accessed using a forall loop of the form

forall *oid* in *cluster* [*suchthat-clause*] [*by-clause*] *statement*

Except for the inclusion of objects in derived clusters, the semantics of the forall loop are the same as those of the for loop for iterating over a cluster. Thus, given the class item and the derived class stockitem as defined earlier, the statement

```
forall ip in item
    tot_wt += ip->weight_kg;
```

computes the weight of all items including stockitems.

5. RELATED WORK

Many research efforts have attempted to add the notion of persistence and database-oriented constructs in programming languages (see, for example, [5, 6, 9, 13, 14, 19-23, 25-27, 30, 33]). Some of these database programming languages have been surveyed and compared in [7]. Issues in integrating databases and programming languages have been discussed in [7, 10, 11, 23, 29, 34]. In the remainder of this section, we limit our discussion to some other ongoing work on combining C and C++ with databases.

Closely related to our work is the language E [23, 24], which also started with C++ and added persistence to it. However, persistent objects in E must be of special types called "db" types. Objects of such types can be volatile or persistent. Persistence orthogonality in E can thus be realized by programming exclusively in "db" types, but all references to volatile objects in that case would incur run time check to see if they need to be read into memory [23]. Otherwise, one has to have two class definitions, one "db" type and one "non-db" type, if objects of the same type are to be allocated both in volatile memory and in persistent store.

Other efforts to extend C and C++ to provide persistence are Avalon/C++ [15], Vbase [6] which combines an object model with C, the O2 system [17] which also integrates an object model with C. More discussion about E and the above efforts can be found in [3].

6. FINAL COMMENTS

An important goal of research in programming languages design is to provide a better fit between the application domain and the programming notation. We started with the object-oriented facilities of C++ and extended them with features to support the needs of databases, putting to good use all the lessons learned in implementing today's database systems. At the same time, we tried to maintain the spirit of C++ (and C) by adding only those facilities that we considered essential for making O++ a database programming language. We feel that O++ provides a clean fusion of database concepts in an object-oriented programming language.

REFERENCES

[1] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[2] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[3] R. Agrawal and N. H. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", *2nd Int'l Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[4] A. V. Aho and J. D. Ullman, "Universality of Data Retrieval Languages", *Proc. 6th ACM Symp. Principles of Programming Languages*, San-Antoniao, Texas, Jan. 1979, 110-120.

[5] A. Albano, L. Cardelli and R. Orsini, "Galileo: A Strongly Typed Interactive Conceptual Language", *ACM Trans. Database Syst. 10*, 2 (June 1985), 230-260.

[6] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented development Environment", *Proc. OOPSLA '87*, Orlando, Florida, Oct. 1987, 430-440.

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

[8] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 16-52.

[9] F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, "FAD, a Powerful and Simple Database Language", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 97-105.

[10] T. Bloom and S. B. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages", *Proc. OOPSLA '87*, Orlando, Florida, Oct. 1987, 441-451.

[11] P. Buneman, "Can We Reconcile Programming Languages and Databases?", in *Databases – Role and Structure: An Advanced Course*, Cambridge Univ. Press, Cambridge, England, 1984, 225-243.

[12] P. Buneman and M. Atkinson, "Inheritance and Persistence in Database Programming Languages", *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 4-15.

[13] L. Cardelli, "Amber", LNCS 242, 1986.

[14] G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Massachusetts, June 1984, 316-325.

[15] D. D. Detlefs, M. P. Herlihy and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *IEEE Computer 21*, 12 (Dec. 1988), 57-69.

[16] S. N. Khoshafian, G. P. Copeland and 406-416, "Object Identity", *Proc. OOPSLA '86*, Portland, Oregon, Sept. 1986.

[17] C. Lecluse, P. Richard and F. Velez, "O_2, an Object-Oriented Data Model", *Proc. ACM-SIGMOD 1988 Int'l Conf. on Management of Data*, Chicago, Illinois, June 1988, 424-433.

[18] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research", Tech. Rep.-88-001, Int'l Computer Science Inst., Berkeley, California, May 1988.

[19] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulis and M. Stanley, "Implementation of a Compiler for a Semantic Data Model", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 118-131.

[20] P. O'Brien, P. Bullis and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proc. Int'l Workshop Object-Oriented Database System*, Asilomar, California, Sept. 1986, 113-123.

[21] Persistent Programming Research Group, "The PS-Algol Reference Manual, 2d ed.", Tech. Rep. PPR-12-85, Computing Science Dept., Univ. Glasgow, Glasgow, Scotland, 1985.

[22] J. E. Richardson and M. J. Carey, "Programming Constructs for Database System Implementation in EXODUS", *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Fransisco, California, May 1987, 208-219.

[23] J. E. Richardson and M. J. Carey, "Persistence in the E Language: Issues and Implementation", Computer Sciences Tech. Rep. #791, Univ. Wisconsin, Madison, Sept. 1988.

[24] J. E. Richardson, M. J. Carey and D. H. Schuh, "The Design of the E Programming Language", Computer Sciences Tech. Rep. #824, Univ. Wisconsin, Madison, Feb. 1989.

[25] L. Rowe and K. Shoens, "Data Abstraction, Views and Updates in RIGEL", *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, Boston, Massachusetts, May-June 1979, 77-81.

[26] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl and R. Meyer, "OOPS – An Object Oriented Programming System with Integrated Data Management Facility", *Proc. IEEE 4th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 118-125.

[27] J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247-261.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System", *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, May 1979, 23-34.

[29] M. Shaw and W. A. Wulf, "Toward Relaxing Assumptions in Languages and their Implementations", *SIGPLAN Notices Notices 15*, 3 (March 1980), 45-61.

[30] J. M. Smith, S. Fox and T. Landers, *ADAPLEX: Rationale and Reference Manual, 2nd ed.*, Computer Corp. America, Cambridge, Mass., 1983.

[31] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley., 1986.

[32] B. Stroustrup, "Multiple Inheritance for C++", *Proc. EUUG conference*, Helsinki, May 1987, 189-208.

[33] A. Wasserman, "The Data Management Facilities of PLAIN", *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, Boston, Massachusetts, May-June 1979.

[34] S. N. Zilles, "Types, Algebras and Modeling", *SIGPLAN Notices Notices 16*, 1 (Jan. 1981), 207-209.

An Object-Oriented Query Algebra[†]

Gail M. Shaw and Stanley B. Zdonik[‡]

Department of Computer Science
Brown University
Providence, R.I. 02912

Abstract

We define a query algebra for object-oriented databases that fully supports abstract data types and object identity while providing associative access to objects, including a join capability that respects the discipline of data abstraction. The structure of the algebra and the abstract access to objects offer opportunities for query optimization.

The algebraic operations take an abstract view of objects and access typed collections of objects only through the public interface defined for the type. The algebra supports access to relationships implied by the structure of the objects, as well as the definition and creation of new relationships between objects. We introduce two notions of object equality to support the creation of new objects by the algebraic operations.

1 Introduction

We are interested in efficiently accessing data in an object-oriented database. Relational database systems offer efficient access to large amounts of data but place the first normal form restriction on the structure of the data. This restriction supports the development of query languages and optimization strategies for that model.

The algebra presented in the next section synthesizes relational query concepts with object-oriented databases (see also [Sha89b]). The algebra supports an object-oriented model with abstract data types, encapsulation, type inheritance, and object identity. Unlike other languages proposed for object-oriented databases (e.g. [Zan83], [Ban87], [Mai87], [Ban88], [Car88], [Osb88]) our algebra fully supports these object-oriented concepts and still provides full associative access to the database, including a join capability that respects the discipline of data abstraction. In addition, the algebra is strongly-typed and can be statically type-checked.

We consider the type of every object to be an abstract data type. The implementation of objects, and their attributes and behavior, are invisible to the query algebra, and all access to objects is through the interface defined for the type. The types are structured as an inheritance hierarchy. The only restriction the algebra places on type inheritance is substitutability; an object of type A can be used in any context expecting a supertype of A. We distinguish between type and collections of objects having that type, similarly to GemStone[Mai87] and EXTRA[Car88], and query over the collections using the type structure as the database scheme. The collections are treated as though they are homogeneous; the member type associated with a collection can be a supertype of the types of the objects in the collection. The result of a query is a new, typed database collection, with the member type of the collection determined statically using the type structure.

Our support for object identity leads to a model and algebra in which queries can build new strongly-typed objects, duplication in set membership is distinguished using objects' identity, comparisons between objects can distinguish between value equality and identifier equality, and algebraic operations can manipulate identities. Theoretical aspects of support for identity are addressed in the query language IQL [Abi89].

[†] A fuller version of this paper appears in the Proceedings of the 2d International Workshop on Database Programming Languages [Sha89a].

[‡] Support for this research is provided by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under contract N0014-88-K-0406, by Apple Computer, Inc., and by US West.

Our algebra addresses many of the same concerns as IQL, and additionally illustrates that data abstraction can be supported along with identity.

Many systems make the distinction between value equality and identifier equality for objects by providing operators testing identity, deep equality and, possibly, shallow equality (e.g. [Ban87], [Lec88], [Osb88], [Car88]). Two objects are identical when they are the same object, shallow equal when they have the same value, and deep equal when they have the same type and a recursive traversal of the objects eventually finds identical values (see [Kho86]). Our algebra supports the semantics of these operations by assuming that the equality operator refers to object identity and using an extended notion of deep equality (see Section 3).

Identity is also an issue when returning objects satisfying query predicates. We use object identity as a test for set membership, as do most other systems, but we also allow, to a limited extent, the introduction of deep-equality semantics in determining set membership. The need for deep-equality results from the ability to create new, strongly-typed objects in response to queries.

Many object-oriented database systems (e.g. [Mai87], [Alb85], [Ban88]) only support selection of objects already existing in the database. We recognize that many relationships requested by queries will exist in the database objects, but also note that existing objects in the database may not necessarily reflect all relationships requested by a query. Thus, we additionally provide operators to create new relationships. The algebra can be used to join objects from different collections by creating tuples to store relationships between objects in those collections. These tuples are new objects with well-defined types and unique identities. The creation of these tuples as statically-typed objects is supported by the existence of parameterized types in the data model. Our algebra can simulate the join of [Car88] and the unnest join of [Kor88], but not the combine operation of [Osb88] since the latter does not necessarily maintain the integrity of abstract data types.

A major aspect in the development of query languages is the potential for optimization. Query algebras can support optimization through syntactic transformations (e.g., [Ban87], [Osb88]). However, the implementation of abstract data types involves the introduction of new operations, making optimization more complicated [Zdo89]. Syntactic transformations are not sufficient for such systems, and may need to be combined with optimization strategies for encapsulated behaviors (e.g. [Gra88]). We expect the structure of our algebra and our consistent approach to abstract data types to offer opportunities for combining such optimization strategies.

In the next section we discuss the Encore object-oriented data model and present our algebra for querying under that model. The algebraic support for object identity is examined in section 3. This support leads to special operators for the manipulation of objects and new definitions for object equality, which are also presented in that section. The algebra forms a basis for our current research into optimization of object-oriented queries and we introduce that research in section 4.

2 The Encore data model and query algebra

The data model for our query algebra is based on the Encore object-oriented data model [Zdo86]. The model includes abstract data types, type inheritance, typed collections of typed objects, and objects with identity. We query over collections of objects using the type of objects in the collection as a scheme for the collection. The collections are considered to be homogeneous, although the objects in the collection may have a type which is a subtype of the member type of the collection. We assume subtyping supports substitutability: if T is a subtype of S then an instance of T can be used in any context expecting S.

A type is an abstract data type, and consists of an interface and implementation. Queries are concerned with the type interface, although their optimization may be concerned with the type implementation. An abstract type definition includes the *Name* of the type, a set of *Supertypes* for the type, a set of *Properties* defined for instances of the type and a set of *Operations* which can be applied to instances of the type.

Properties reflect the state of an object while operations may perform arbitrary actions. Properties are typed objects that may be implemented as stored values, procedures or functions. We assume the implementation of a property returns an object of the correct type and has no side-effects. The query algebra treats properties as stored values, addressing a property using dot-notation (e.g. $s.q$ where s is an object of type T and q is a property of T). If necessary, a request for a property will cause invocation of a function implementing the property, resulting in the return of an object representing the requested property of the object.

In addition to user-defined abstract data types, we assume a collection of atomic types (Int, String,

etc.), a global supertype Object, and parameterized types Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$] and Set[T].[1] Type Object defines equality properties for all objects. Property Equal ($=$) refers to the identical comparison for objects. We also assume, for now, that shallow-equal ($=_s$) and deep-equal ($=_d$) [Kho86] are available for object comparisons. Other comparisons may be user-defined for sub-types of Object.

The parameterized types have fixed sets of properties and operations, and they allow the creation of new, strongly-typed objects. The Tuple type associates types (T_i) for attributes (A_i) and defines property *Get_attribute_value* and operation *Set_attribute_value* for each attribute. Parameterized type Set[T] declares T as the type, or supertype, of objects in a collection having type Set, and defines operations *in* and *subset_of* (among others). Duplication in set membership depends on object identity; a set will not contain two objects with the same identifier.

The query algebra provides type specific operations against collections of encapsulated objects with identity. The algebraic operations support the notion of abstract data type, and all access to objects in a collection is through the public interface defined for the collection member type. The operations are also concerned with object identity; new objects with unique identities can be created and we provide operators to manipulate the identities of objects.

We define the algebraic operations:

$$\begin{aligned} Select(S,p) &= \{s \mid (s\ in\ S) \wedge p(s)\} \\ Image(S,f) &= \{f(s) \mid s\ in\ S\} \\ Project(S, < (A_1,f_1),...,(A_n,f_n) >) &= \\ &\quad \{< A_1 : f_1(s),...,A_n : f_n(s) > \mid s\ in\ S\} \\ Ojoin(S,R,A,B,p) &= \{< A : s, B : r > \mid s\ in\ S \wedge r\ in\ R \wedge p(s,r)\} \end{aligned}$$

where S and R are collections of objects[2], p is a first-order Boolean predicate, the A_i's are names for attributes whose values are objects of type T_i, and the f_i's are functions returning objects of type T_i. We also define standard set operations *Union*, *Difference* and *Intersection*, formatting types of operations *Flatten* (for a set of sets), *Nest* and *UnNest* (as in [Jae82], with object identifiers as attribute values), and operators *DupEliminate* and *Coalesce* to manage identity of objects.

The *Select* operation creates a collection of database objects satisfying a selection predicate. This type of operation, which returns existing database objects, is standard for querying. We recognize that many relationships requested by queries will exist in the database objects, but also note that existing objects in the database may not necessarily reflect all relationships requested by a query. In addition, the relationship implied by the presence of objects in a particular collection may not be the relationship from which a query wishes to select. Thus, we provide operators *Image*, *Project*, and *Ojoin* to handle relationships not defined by an object type.

The *Image* operation describes the application of a function to each object in the queried collection. In some ways, this is a selection of objects from a different collection than the one over which the query is defined. For example, suppose we have a collection Stacks containing stacks of books. The query $Image(Stacks, \lambda s\ s.top)$ returns a set of objects representing the books that are at the top of some stack. This selection is not done, however, from the collections of the books themselves.

The *Project* operation extends Image by allowing the application of more than one function to an object, thus supporting the maintenance of selected relationships between properties of an object. Project can also be used to build relationships between objects in different collections. Project returns one tuple for each object in the collection being queried, and all tuples have unique identifiers. The tuples have type Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$], where T_i is the return type of f_i. All properties and operations for type Tuple are thus defined for the objects of the result set.

Functions for Project and Image are commonly built by composing the application of properties of objects. For example, $Image(Stacks, \lambda s\ s.top.title)$ returns a set of objects representing the titles of books at the top of some stack in the collection. A function, however, may be any well-defined operation. For example, consider the query

$$Project(Stacks, \lambda s < (S,s), (Len, s.length), (Q, Select(Queues, \lambda q\ q.length = s.length)) >)$$

[1] Other parameterized types might be useful for forming collections of objects, but we have not yet addressed the semantics of our algebra on such types of collections. For now, if we want to apply the algebraic operations to other types of collections, the collection would first have to be coerced into a set.

[2] We assume in this definition of Ojoin that S and R are collections of non-tuple objects, and discuss joins involving collections of tuples later. For the other operations, the type of objects in S or R is arbitrary.

which returns, for each stack in the collection, a tuple containing the stack, its length, and a set of all queues having the same length as the stack. The length property is defined for each stack and, since Queues is a collection, the Select operation is defined over Queues. All properties and operations for type Tuple are defined on the result set. In this example the only valid operations on the result tuples are *get_S*, *get_Len*, and *get_Q*. However, for any tuple t in the result, all stack operations can be applied to $t.S$ and all set operations to $t.Q$.

The *Ojoin* operator is an explicit join operator used to create relationships between objects from two collections in the database. The operation creates new tuples in the database to store the generated relationships. The tuples are strongly typed using the parameterized type Tuple, with each attribute typed according to the type of the collection from which it is derived. Each tuple has a unique identity. The objects involved in the relationships are maintained and can be accessed as the value of the appropriate attribute in the tuple.

We assumed, in the Ojoin definition given, that S and R are both collections of objects having abstract data types (not type Tuple). However, if S, for example, is a collection of n-tuples, Ojoin creates (n+1)-tuples (n attributes from S and one attribute to hold an object from R)[3]. This definition of Ojoin preserves the associativity of the operation, which we expect to be useful in query optimization.

For example, suppose we have set Stacks, as before, and a set Queues, containing queues of books. Then $SQ := Ojoin(Stacks, Queues, S, Q, \lambda s \lambda q\ s.Top = q.Front)$ returns a set of stack-queue pairs in which the top of the stack is identical to the front element of the queue. If we then join this result with a set Lists containing lists of books (i.e. $Ojoin(SQ, Lists, , L, \lambda t \lambda l\ t.S.Top = l.First)$) the result is a collection of 3-tuples with attributes S, Q, and L. The only operations available on the result tuples are those operations defined for tuples (properties get_S, get_Q, and get_L). However, for each element t in the result, all stack operations are available for $t.S$, queue operations for $t.Q$, and list operations for $t.L$.

Union, *Difference* and *Intersection* are the usual set operations with set membership based on object identity. The result for all operations is considered to be a collection of objects of type T, where T is the most specific common supertype (in the type lattice) of the member types of the operands. Any two sets can be combined, since all types have a common supertype of Object. The types of the objects themselves are not changed; the type of each object will be the member type, or a subtype of the member type, of the result collection. However, the only operations available to subsequent queries against the result collection will be those defined for the result member type.

The *Flatten* operation is used to restructure sets of sets; it takes an object of type Set[Set[T]] and returns an object of type Set[T]. This operation is useful in conjunction with Image. Image can extract components with type set from an object; Flatten allows consideration of the set members individually.

Nest and *UnNest* extend the same operators for non-first normal form relations (see [Jae82]) to sets of objects with identity. Sets of tuples can be unnested on a single set-valued attribute, or nested to create a set-valued attribute. The UnNest operation creates a new tuple for each object in the designated attribute. For example, a tuple $t =< A : o_1, B : s_1 >$, where object s_1 is a set containing objects o_2 and o_3, is unnested on the B attribute to create two new tuples: $t_1 =< A : o_1, B : o_2 >$ and $t_2 =< A : o_1, B : o_3 >$. Conversely, a Nest on the B attribute of a set containing tuples t_1 and t_2 would create a tuple shallow-equal to t.

3 Object identity and equality

The result of a query using the algebra presented thus far is a new collection of objects. The collection will be a newly identified object in the database, and thus there cannot be identical responses to a query. The objects in a result collection may be either existing database objects or new objects created during the operation. The creation of new objects means we may be creating multiple objects with unique identifiers when a single object is desired. The new objects will often be shallow-equal, although nested operations could create results that are deep-equal. For example, a Select inside a Project would create new collection objects inside new tuple objects. The nested objects illustrate a need for a refined notion of object equality, since deep-equality compares not only the new objects built by the query but all properties of those objects (i.e., eventually the database objects).

We define *i-equality*, where i indicates how "deep" a comparison should go when examining equality. Identical objects are *0-equal* ($=_0$) and, for $i > 0$, two objects are *i-equal* ($=_i$) if they are both collections of

[3] Ojoining with collections of tuples is similar to cartesian product (with selection) followed by tuple collapse of [Abi88].

the same cardinality and there is a one-to-one correspondence between the collections such that corresponding members are $=_{i-1}$, or they both have the same type (not collection) and the values of corresponding properties are $=_{i-1}$. We now use the term shallow-equal to refer to $=_1$; deep-equal ($=_d$) refers to i-equality for some i.

We augment the algebra with operations *DupEliminate* and *Coalesce* to handle situations where i-equal objects are created by a query . Operation *DupEliminate(S,i)* keeps only one copy of i-equal objects from collection S. Such duplication of objects can be created by operations Image, Project and UnNest. Project and UnNest create newly-identified tuples, by definition, and a function applied by the Image operation may create new objects.

For example, suppose we want to group the stack objects in collection Stacks by length. One way to do this would be to assemble, for each stack in Stacks, a collection of all stacks with the same length as follows:

$$Image(Stacks, \lambda j\ Select(Stacks, \lambda s\ s.length = j.length)) \quad (1)$$

However, this result is not exactly what was desired; there may be many collections containing exactly the same stacks, i.e. many shallow-equal collections. If there are x stacks of a particular length, there will be x collections all containing the same x stack objects. However, if DupEliminate is applied to the result as follows

$$DupEliminate(Image(Stacks, \lambda j\ Select(Stacks, \lambda s\ s.length = j.length)), 1) \quad (2)$$

the result would contain the expected collections of stacks with no shallow-equal (1-equal) duplication.

Duplication can be introduced at any level by the algebraic operations. For example, consider an extension to example 1, where we want to store the length along with the set of stack objects having that length:

$$Project(Stacks, \lambda j < (Len, j.length), (S, Select(Stacks, \lambda s\ s.length = j.length)) >) \quad (3)$$

The result of this query will be a set of tuples, one for each object in Stacks. Many of those tuples will have duplicate Len values, and those that are duplicates will have S attribute values that are shallow-equal. In this case, tuples which would be considered to be duplicates will not be shallow-equal, but will be 2-equal.

Query operations Project, UnNest and Image can be modified with $=_i$ as a shorthand for application of operation DupEliminate after application of the operation. Problem 2 could be written, for example, using $Image_{=1}$, and duplicates could have been eliminated from query 3 by using $Project_{=2}$. It is interesting to note that the application of operation DupEliminate can eliminate duplication created by the algebraic operations, as in the preceeding examples, but may also be used to eliminate duplication inherent in the data.

Operations such as Project (as in example 3) and Nest can create tuples with *components* (attribute values) that are i-equal. We define operation $Coalesce(S, A_k, i)$ which, for collection S of tuple objects, eliminates i-equal duplication in the A_k components of the tuples. If two (or more) tuples in S, call them t_1 and t_2, are such that $t_1.A_k =_i t_2.A_k$, then one of the objects is selected (say $t_1.A_k$) and replaces the corresponding attribute values in the other tuples (i.e., the A_k attribute of t_2 is assigned the object $t_1.A_k$).

For example, when *Nest*ing a collection on a given attribute; a new set is created for each tuple in the result, and each new set is a distinct object. However, some of these new set objects may be shallow-equal. Consider the four tuples:

$$t_1 = < A : o_1, B : o_2 > \qquad t_2 = < A : o_3, B : o_4 >$$
$$t_3 = < A : o_1, B : o_4 > \qquad t_4 = < A : o_3, B : o_2 >$$

and the operation $Nest(S, B)$. This operation will result in the creation of two set objects (s_1 and s_2), two tuple objects (t_5 and t_6) and the result R as follows:

$$s_1 = \{o_2, o_4\} \quad and \quad s_2 = \{o_2, o_4\}$$

$$t_5 = < A : o_1, B : s_1 > \quad and \quad t_6 = < A : o_3, B : s_2 >$$

$$R = \{t_5, t_6\}$$

The duplication in objects s_1 and s_2 is similar to that created by the nested Select in problem 3, however operation DupEliminate does not apply here since the A attribute values are distinct in the result tuples.

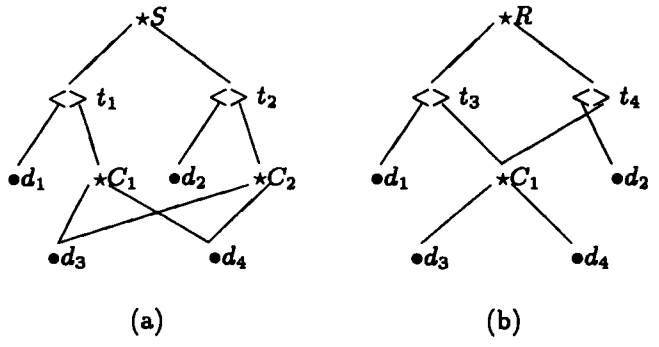

Figure 1: Graphs of collection objects S and R.

We could, however, execute $Coalesce(R, B, 1)$ to modify R. The Coalesce operation would eliminate s_2, and modify t_6 to have value $< A : o_3, B : s_1 >$.

As with operation DupEliminate, the application of Coalesce could eliminate duplication inherent in the data as well as duplication created by the algebraic operations. Project and Nest can create results with duplicate components, as noted above, and any application of Coalesce at a level deeper than that created by Project or Nest would eliminate duplication that was present in the data before application of the algebraic operation.

We also note that i-equality is not sufficient to distinguish between objects that have, and have not, been Coalesced. In order to detect the similarities in data, i-equality ignores the identities of the objects in which the data is stored. We define a stronger notion of equality, called *id-equality*, which can detect structural differences in objects.

Objects can be represented graphically, with nodes labelled with object identifiers or atomic values, arcs connecting collection type objects to all objects in the collection, and arcs also connecting non-collection type objects to the values of all properties of those objects. For example, in Figure 1, S is a collection of tuples containing objects t_1 and t_2. Each tuple in the collection has two attributes, call them A and B. Attribute A has an abstract data type and attribute B has type collection. C_1 and C_2 are collection objects, and d_1 through d_4 are database objects having some abstract data type.

We define two objects to be *id-equal at depth i* if they are i-equal and their graphical representations are isomorphic. I-equality is necessary, but not sufficient, to ensure id-equality. For example, in Figure 1, objects S and R are 3-equal; $C_1 =_1 C_2$ implies that $t_2 =_2 t_4$, thus $S =_3 R$. However, they are not id-equal at any depth. In particular, in R both t_3 and t_4 have identical values for their B attribute (i.e. C_1) but in S the B attributes of t_1 and t_2 are 1-equal.

Id-equality and i-equality differ in their treatment of object identity. Identities are transparent to i-equality comparisons. This allows the comparison of objects when identity is not important, for example in duplicate removal. Id-equality, on the other hand, retains some of the semantics of the data implied by the identity of the objects. In particular, id-equality recognizes aliasing of properties of objects. In Figure 1, for example, $t_3.B$ and $t_4.B$ refer to the same object (C_1), thus modification to t_3 in R can affect t_4. S and R are not id-equivalent because the tuples in S can behave differently than the tuples in R. In particular, modification to $t_3.B$ affects t_4 in R while, in S, t_2 is independent of modifications to t_1. The support for both definitions for equality in the model and algebra allows the user to determine the extent to which object identity is needed in an application.

4 Implications for Query Optimization

We expect the structure of the operations and the redundancies in the operator set presented here to offer many opportunities for optimization. Query optimization may also depend on the implementations of the types queried, but we do not consider that aspect yet.

The similarities between our algebra and relational algebra imply that relational optimization results may prove useful in object-oriented query optimization. For example, consider the transformation rules in

$$Select(Select(S, p_1), p_2) = Select(Select(S, p_2), p_1) \quad (1)$$

$$Select(Select(S, p_1), p_2) = Select(S, p_1 \wedge p_2) \quad (2)$$

$$Union(Select(S, p_1), Select(S, p_2)) = Select(S, p_1 \vee p_2) \quad (3)$$

$$Union(Image(S_1, f), Image(S_2, f)) = Image(Union(S_1, S_2,)f)) \quad (4)$$

$$Select(Ojoin(A, B, p), \lambda t\ p_s(t.A)) = Ojoin(Select(A, \lambda a\ p_s(a)), B, p) \quad (5)$$

$$Select(Ojoin(A, B, \lambda a \lambda b\ p(a, b)), \lambda t\ p_s(t.A, t.B)) = Ojoin(A, B, \lambda a \lambda b\ p(a, b) \wedge p_s(a, b)) \quad (6)$$

$$Ojoin(A, B, \lambda a \lambda b\ p(a) \wedge p'(a, b)) = Ojoin(Select(A, \lambda a\ p(a)), B, \lambda a \lambda b\ p'(a, b)) \quad (7)$$

$$Select(S_1, \lambda s\ s.a\ in\ S_2) = Image(S_2, \lambda a\ inverse(a)) \quad (8)$$

$$Select(S_1, \lambda s\ s.a_1.a_2\ in\ S_3) = Select(S_1, \lambda s\ s.a_1\ in\ Select(S_2, \lambda s\ s.a_2\ in\ S_3)) \quad (9)$$

Table 1: Some Algebraic Identities

Table 1. Selection predicates can be applied in any order (identity 1) and can be combined (identities 2 and 3), as in relational algebra. Identity 5 is equivalent to the relational optimization strategy of pushing Selection past Join. Similarly, when a Select operation is composed with an Ojoin, it may be possible to instead compose the two predicates to produce a single operation (identity 6). These two ideas are combined in identity 7; if an Ojoin predicate contains a conjunct involving only one of the operands, that conjunct can be extracted from the Ojoin predicate to form a Select predicate on the appropriate operand.

We can also take advantage of knowledge about the database types. For example, consider the following query that retrieves the departments whose managers are ready to retire:

$$Select(Departments, \lambda d\ d.Manager\ in\ RetireThisYear)$$

This query relies on the fact that type Department has a Manager property and RetireThisYear is a collection of managers. If we also know that each Manager also has a Department property (i.e. Dept of Manager is an inverse of Manager of Dept) rule 8 could be applied to produce the following equivalent query:

$$Image(RetireThisYear, \lambda m\ m.Dept)$$

The Image will extract the Dept, i.e. inverse, property for each manager in the RetireThisYear set. This optimization technique relies on the fact that the inverse property exists for the property that was originally used, i.e. two properties a_1 and a_2 are defined to obey the constraint $(x.a_1 = y) \Leftrightarrow (y.a_2 = x)$. Although this is not true in the general case, if the collections in the query are relations, the computation of the property values and their inverses is all done with a single application of a join operation, which is inherently bidirectional.

The redundancies present in the operator set may also offer opportunities for optimization. For example, Ojoin is equivalent to the composition of Select, UnNest and Project as follows:

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a, b)) = Select(UnNest(Project(As, \lambda a < (A, a), (B, Bs) >), B), \lambda a \lambda b\ p(a, b))$$

where, if As and Bs are collections of objects of type T_A and T_B, respectively, the result is a collection of tuples with attributes A and B having types T_A and T_B, respectively. Other redundant operations include Flatten (use Project, UnNest and Image), UnNest(use Project, Image and Flatten), Intersection(use Difference), Nest (use Project, Image and Select), and Coalesce (use Image, Ojoin and Project).

5 Summary

The algebra presented here respects the encapsulation and strong typing rules of our object-oriented database system, and at the same time provides the ability to construct dynamic relationships. The query algebra supports abstract data types by accessing objects only through the interface defined by their type. Results of queries are collections of existing objects or collections of tuples, built by the query. The creation of collections and tuples with statically determined types is supported by the existence of parameterized types in the model.

The algebra also supports object identity by managing the creation of new objects and by considering object identity when manipulating these objects. Support for object identity leads to a need for new definitions for equality of objects. I-equality compares objects based on their type and the values of their properties. Id-equality extends this definition to also consider, for i-equal objects, the structures implied by the identifiers associated with their property values.

Our current research involves the optimization of queries using the algebra presented in this paper. The structure of the operations and the operator set offer many opportunities for optimizations that are independent of the implementation of the types queried. We expect to combine those with type-dependent optimizations to provide efficient query access to data in our object-oriented database. We are also investigating techniques for estimating the cost of query expressions. The encapsulation provided by the object model complicates this problem.

References

[Abi88] Serge Abiteboul and Catriel Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical Report No. 846, INRIA, 1988.

[Abi89] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In *SIGMOD Proceedings*. ACM, 1989.

[Alb85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *Communications of the ACM*, 10(2):230–260, June 1985.

[Ban87] Francois Bancilhon et al. FAD, a Powerful and Simple Database Language. In *Proceedings of the 13th VLDB Conference*, pages 97–105, 1987.

[Ban88] Jay Banerjee, Won Kim, and Kyung-Chang Kim. Queries in Object-Oriented Databases. In *Proceedings 4th Intl. Conf. on Data Engineering*, pages 31–38. IEEE, Feb 1988.

[Car88] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg. A Data Model and Query Language for EXODUS. In *SIGMOD Proceedings*, pages 413–423. ACM, June 1988.

[Gra88] Goetz Graefe and David Maier. Query Optimization in Object-Oriented Database Systems: A Prospectus. In *Advances in Object-Oriented Database Systems*, pages 358–363. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Jae82] G. Jaeschke and H. J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the Symposium on Principles of Database Systems*, pages 124–138. ACM, March 1982.

[Kho86] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416. ACM, September 1986.

[Kor88] Henry F. Korth. Optimization of Object-Retrieval Queries. In *Advances in Object-Oriented Database Systems*, pages 352–357. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Lec88] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *SIGMOD Proceedings*, pages 424–433. ACM, June 1988.

[Mai87] David Maier and Jacob Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[Osb88] S. L. Osborn. Identity, Equality and Query Optimization. In *Advances in Object-Oriented Database Systems*, pages 346–351. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Sha89a] Gail M. Shaw and Stanley B. Zdonik. An Object-Oriented Query Algebra. In *2nd International Workshop on Database Programming Languages*, June 1989.

[Sha89b] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. Technical Report CS-89-19, Brown University, 1989.

[Zan83] Carlo Zaniolo. The Database Language GEM. In *SIGMOD Proceedings*, pages 207–218. ACM, May 1983.

[Zdo86] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.

[Zdo89] Stanley B. Zdonik. Query Optimization in Object-Oriented Database Systems. In *Proceedings of the Hawaii International Conference on System Science*, January 1989.

HiLog as a Platform for Database Languages*

(or why predicate calculus is not enough)

- Extended Abstract -

Weidong Chen Michael Kifer David S. Warren

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract

We argue that predicate calculus is not sufficient as a basis for the next generation of database languages. To fill in the gap, we propose a novel logic, called HiLog, which can be viewed as an extension of predicate calculus. The distinctive feature of HiLog is its higher-order syntax which makes it possible to manipulate the database with greater ease. However, the semantics of this logic is essentially first-order, which makes it possible to define a resolution-based proof procedure for HiLog. We then go on and compare HiLog with two other well-known database languages, COL and LDL, arguing that HiLog eliminates some of the problems in these languages caused by their second-order semantics. Finally, we discuss the utility of HiLog as a platform for implementing object-oriented database languages.

1 Preface

Manipulating predicates, functions, and even atomic formulas is a commonplace in logic programming. For example, Prolog combines predicate calculus, higher-order and meta-level programming in one working system, allowing programmers routine use of generic predicate definitions (e.g. transitive closure, sorting) in which predicates can be passed as parameters and returned as values [8]. Another well-known non-first-order feature is the "call" meta-predicate of Prolog. Usefulness of higher-order constructs in the database context has been pointed out in many works, including [17,19,29].

Although Prolog is based on first-order predicate calculus, the latter does not have the wherewithal to support any of the aforesaid features, and consequently they have an ad hoc status in logic programming. In this paper, we investigate the fundamental principles underlying higher-order logic programming and, in particular, shed new light on why and how these Prolog features appear to work in practice. We propose a novel logic, called HiLog, which provides a clean declarative semantics to much of this higher-order logic programming.

From the outset, even the terminology of "higher-orderness" seems ill-defined. A number of works have proposed various higher-order constructs in the logic framework [1,5,10,8,14,17,20,21,30,33] but with such a diversity of syntax and semantics, it is not always clear what kind of higher-orderness is being claimed. In our opinion, there are at least two different facets to the issue: a higher-order syntax and a higher-order semantics. Logicians seem to have understood this separation for quite some time and, for example, in [12] a first-order *semantics* is described for the *syntactically* second-order predicate calculus.

Informally, by higher-order *syntax*, logicians mean a language in which variables are allowed to appear in places where normally predicate and/or function symbols do. In contrast, higher-order *semantics* is

*This is an extended abstract of the paper with the same title that appeared in the Proceedings of the 2-nd Intl. Workshop on Database Programming Languages, R. Hull, R. Morison, D. Stemple (eds.), Morgan Kaufmann Publ., Sept. 1989

manifested by semantic structures in which variables may range over domains of relations and functions constructed out of the domains of individuals. In first-order semantic structures, variables can only range over domains of individuals. It should be noted that the classification based upon semantics has no implication whatsoever regarding the "intrinsic higher-orderness". It is quite possible to replace higher-order semantics for some languages by an entailment-equivalent first-order semantics. On the other hand, it is well known that some semantics (e.g. the standard semantics of second order predicate calculus) are inherently higher-order and no equivalent first order substitute exists for the corresponding languages.

According to the above classification, we can group higher-order logics into four categories, and some of the logic systems are listed in the following table.

	Syntax	
Semantics	First-Order	Higher-Order
First-Order	first-order predicate calculus [12] Definite clause programs[4,35] O-Logic[18,25] C-Logic[9]	λ-Prolog[30,33] F-Logic[17] HiLog
Higher-Order	stratified logic programs[3,24,34] LPS[20,21] LDL[5] LDM[22]	type theory[10,14] (both standard and Henkin's semantics) COL[1] module theory[8]

Notice that stratified logic programming emerges from this classification as a first-order language with a higher-order semantics. This is because the semantics of stratified logic programs is given with the aid of a second-order circumscriptive axiom [24].

In this paper, we present a simple logical framework in which predicates, functions, and atomic formulas can be manipulated as first-class objects. It is quite clear that in order to support such manipulation naturally enough, the syntax has to be higher-order. As explained earlier, switching over to such a syntax leaves open two possibilities for the semantics. Under higher-order semantics, predicates and functions are identified by their extensions, i.e., say, a pair of predicates represents the same thing if and only if their extensions coincide in every appropriate semantic structure. Unfortunately, extensions are notoriously difficult to handle in an efficient manner.

In contrast, under first-order semantics, predicates and functions have associated entities, called *intensions*, which can be manipulated directly. Furthermore, depending on the context, intensions may assume different roles acting as relations, functions, or even propositions. For instance, in λ-Prolog [30,33], a predicate symbol is viewed as an expression when it occurs as an argument and is treated as a relation in contexts when it appears with arguments. In F-logic [17], so called id-terms are handled as individuals when they occur as object identities, they are viewed as functions when occurring as object labels, and as sets when representing classes of objects.

As a rule of thumb, first-order semantics for higher-order languages are generally believed to be more tractable than their higher-order counterparts, and this observation motivates our choice for HiLog. The basic idea is to explicitly distinguish between intensional and extensional aspects in the semantics. Intuitively, intensions identify different entities allowing one to "get a handle" on them, while extensions correspond to various roles these entities play in the real world. It has also been argued by Maida and Shapiro [27,26] that knowledge representation is part of the conceptual structure of cognitive agents, and therefore should not (and even cannot) contain extensions. The reason is that cognitive agents do not have direct access to the world, but only to one of its representations. Our approach is in the same spirit and, consequently, extensions of predicates and functions are not available for direct manipulation. On the other hand, intensions of higher-order entities such as predicates and even atomic formulas can be freely manipulated. Their extensions come into the picture only when the respective expressions need to be evaluated. Thus, HiLog combines advantages of higher-order syntax with the simplicity of first-order semantics, benefiting from both.

This paper and [6] are companions, and have sizable overlap covering the semantics and the proof theory of HiLog. The present paper, however, gives a more in-depth discussion of the database aspects of HiLog, while [6] provides a more detailed coverage of logical and logic programming issues.

The rest of the paper is organized as follows. After a brief motivational discussion, we present the formal syntax and semantics of HiLog. Then we show the utility of HiLog as a database language and, in particular, discuss its relationship to COL [1], LDL [5], and the recently proposed object-oriented logics [9,18,17]. We then describe a resolution-based proof theory for HiLog, and then conclude.

2 Syntax and Semantics of HiLog

2.1 Motivation

Prolog syntax is quite flexible, allowing symbols to assume different roles depending upon the context: a symbol may have different arities, it can be viewed as a constant, a function, a predicate, etc. For instance, in the clause

```
r(X) :- p(X, f(a)), q(p(X, f(a)), f(p, b)).
```

symbol f occurs as both a unary and a binary function, and p appears as a binary predicate as well as a binary function symbol. Furthermore, the same syntactic object, p(X, f(a)), is evaluated as an atomic formula in the first literal of the rule body and as an individual term in the second.

While, in the above example, different occurrences of the same symbol can be semantically disambiguated simply by renaming the different occurrences of f and p, this cannot be done in the following rule:

```
p(X, Y) :- q(Y), X.
```

Here individual variable X occurs as a first-order term and as an atomic formula, and renaming its different occurrences will, intuitively, yield a semantically different clause.

The syntax of HiLog resembles that of Prolog in that HiLog logical symbols are arityless and the distinction between predicate, function, and constant symbols is eliminated. Particularly, a HiLog term can be constructed from *any* logical symbol followed by *any* finite number of arguments. Different occurrences of a logical symbol are viewed as semantically the *same object* characterized by the same intension. Associated with such an intension there are several different *extensions* which capture the different roles the symbol may assume in different contexts.

HiLog allows complex terms (not just symbols) to be viewed as functions and predicates. For example, a *generic* transitive closure predicate can be defined as follows:

```
closure(R)(X, Y) :- R(X, Y).
closure(R)(X, Y) :- R(X, Z), closure(R)(Z, Y).
```

where closure is (syntactically) a second-order function which, given any relation R, returns its transitive closure closure(R). Generic definitions can be used in various ways, e.g.,

```
parent(john, bill).
parent(bill, bob).
manager(john, mary).
manager(mary, kathy).
schema(parent).
schema(manager).
john_obeys(X) :- schema(Y), closure(Y)(john, X).
```

will return { bill, mary, kathy, bob } in response to the query :- john_obeys(X), which is the set of john's ancestors and bosses.

In databases, schema browsing is commonplace [29]. In HiLog, browsing can be performed through the same query language as the one used for data retrieval. For instance,

```
relations(Y)(X) :- X(Y,Z).
relations(Z)(X) :- X(Y,Z).
:- relations(john)(X).
```

will return the set of all binary relations mentioning the token *john*. HiLog shares this browsing capability with another recently proposed language, called F-logic [17].

As seen from the earlier examples, HiLog terms are also atomic formulas. In that capacity their semantics is indirectly captured through the truth meaning of terms. For instance, instead of saying that a pair $< a, b >$ is in the relation for a predicate p, we say that the term $p(a, b)$ denotes a *true proposition*. Formal details are provided in the next section.

2.2 Syntax and Semantics

In addition to parentheses, connectives, and quantifiers, the alphabet of a language **L** of HiLog contains a countably infinite set $\mathcal{V}$ of variables and a countable set $\mathcal{S}$ of logical symbols. We assume that $\mathcal{V}$ and $\mathcal{S}$ are disjoint.

The set $\mathcal{T}$ of HiLog *terms* of **L** is a minimal set of strings over the alphabet satisfying the following conditions:

- $\mathcal{V} \cup \mathcal{S} \subseteq \mathcal{T}$;
- If $t, t_1, \ldots, t_n$ are in $\mathcal{T}$, then $t(t_1, \ldots, t_n) \in \mathcal{T}$, where $n \geq 1$.

Notice that according to this definition, terms can be applied to any number of other terms and logical symbols are arityless. A term is also an *atomic formula*. More complex formulas are built from atomic ones in the usual way by means of connectives $\vee$, $\wedge$, $\neg$ and quantifiers $\exists$, $\forall$.

Because of space limitations in this abstract, we do not provide a full account of the semantics of HiLog. Details can be found in the full version of this paper and in [6]. Just to give a flavour of this semantics, we note that Herbrand interpretations can be defined as subsets of $\mathcal{T}$, pretty much in the style of predicate calculus. Then the notion of satisfaction of formulas by interpretations is defined in a standard way.

3 Examples

As explained earlier, a term can be viewed as an individual, a function, or a predicate in different contexts. When functions or predicates are treated as objects, they are manipulated as terms through their intensions; while being applied to arguments they are evaluated as functions or relations through their extensions. By distinguishing between intensional and extensional aspects of functions and predicates, HiLog preserves the advantages of higher-order logic and eliminates the difficulties with extensions introduced by a higher-order semantics.

Since variables may be instantiated to HiLog terms which in turn have propositional meaning, there is no need for the infamous "call" predicate built into Prolog. For the squeamish, the latter is naturally defined in HiLog as

```
call(X) :- X.
```

which has the intended meaning.

Another example is the *maplist* of Lisp which can be defined as either a higher-order predicate

```
maplist(F, [], []).
maplist(F, [X|R], [Y|Z]) :- F(X, Y), maplist(F, R, Z).
```

or as a generic predicate

```
maplist(F)([], []).
maplist(F)([X|R], [Y|Z]) :- F(X, Y), maplist(F)(R, Z).
```

The latter is possible since HiLog allows complex terms such as maplist(F) to appear as predicates. This kind of complex predicate has also been used in COL [1] for modeling complex objects containing sets, and in [8] for a theory of modules in logic programming. In fact, HiLog provides an alternative, first-order, semantics to the theory of modules of [8]. It yields the same result in most situations, but differs from [8] in certain marginal cases where the inherent difference between the intensional treatment of predicates in HiLog and their extensional treatment in [8] becomes essential.

The example in Section 2.1 shows the usefulness of generic view definitions, such as closure, in databases. Generic transitive closure can also be defined in Prolog:

```
closure(R, X, Y) :- C =.. [R, X, Y], call(C).
closure(R, X, Y) :- C =.. [R, X, Z], call(C), closure(R, Z, Y).
```

However, this is obviously inelegant compared to HiLog (see Section 2.1), since this involves both constructing a term out of a list and reflecting this term into an atomic formula using "call". The point of this example is that the lack of theoretical foundations for higher-order constructs in Prolog resulted in an obscure syntax, which partially explains why Prolog programs involving such constructs are notoriously hard to understand.

In natural language processing, there are cases in which generic grammatical rules become necessary. Optionality of a nonterminal symbol and the occurrence of an arbitrary symbol zero, one, or more times are some of the examples [2]. Such rules can be specified as follows (adapted from [2]):

```
option(X) ⟶ X.
option(X) ⟶ [].

seq(X) ⟶ X, seq(X).
seq(X) ⟶ [].
nonempty_seq(X) ⟶ X, seq(X).
```

This can be naturally translated into HiLog using the conventional method of translating DCG grammars into Prolog rules. For instance, the last production is translated into

```
nonempty_seq(X)(Start, End) :- X(Start, S1), seq(X)(S1, End).
```

Unfortunately, the latter is not a Prolog rule, since complex terms are not allowed as predicates. Because of this limitation, special mechanisms for translating such grammars into Prolog are needed, and papers have been written on that subject (e.g. [2]). In contrast, as we have just seen, translation of generic grammars into HiLog is immediate, and does not require special machinery.

As another example, consider relational expressions composed from, say, binary relations connected by the relational operators minus, union, etc. Suppose that the parser has already produced a parse tree (parsing is easy using Horn clauses) of the expression in the form of a term, say, $minus(union(p, q), inter(q, r))$, or similar. As the next step, we would like to write an evaluator for such expressions, which in HiLog looks as follows:

```
minus(P,Q)(X,Y) :- P(X,Y), ¬Q(X,Y).
union(P,Q)(X,Y) :- P(X,Y).
union(P,Q)(X,Y) :- Q(X,Y).
etc.
```

For comparison, we present an analogue of the above program in Prolog. The simplest approach to this problem seems to be to define a translation predicate, tr, which converts parse trees into Prolog goals, and then use call. The rules for tr are as follows:

```
tr( minus(P,Q), X, Y, ( G1 , not (G2)) ) :- tr(P, X, Y, G1), tr(Q, X, Y, G2).
tr( union(P,Q), X, Y, ( G1 ; G2) ) :- tr(P, X, Y, G1), tr(Q, X, Y, G2).
etc.

tr( p, X, Y, p(X,Y) ).
tr( q, X, Y, q(X,Y) ).
etc.
```

The first observation about this Prolog program is that it is clumsy compared to its HiLog counterpart (notice that the arguments X, Y in tr are essential for the program to run correctly). Second, in Prolog, we have to know the alphabet of the language since we have to list all the facts such as tr(p, X, Y, p(X,Y)) (for each predicate symbol) in advance. This is particularly inconvenient in the database environment when the user may create or delete new relations, since the above program must be updated each time.

The ease of writing the above program in HiLog stems from the ability to represent intermediate results of query evaluation in a natural way. For instance, $minus(p, q)$ can be viewed as the name of an intermediate relation for the result of subtracting Q from P. However, it should be clear that in order to take full advantage of HiLog, arities of all relations must be known at compile time, since we must know how many variables should appear in various places in rules. Therefore, rules for the relational operators which do not change the arities of relations (such as the ones above) look particularly attractive in HiLog. On the other hand, operators such as join or Cartesian product require a heavier machinery, such as *functor* and *arg* (which can be formalized in HiLog, by the way [6]). Still, this program looks much more elegant in HiLog than in Prolog.

4 HiLog as a Database Programming Language

In this section, we show that HiLog provides an alternative (first-order) semantics to some of the well-known database languages with higher-order semantics, thereby eliminating some of their problems. Specifically, we focus on COL [1] and LDL [5]. We argue that the first-order semantics for COL and LDL that stems from HiLog is computationally more attractive than the original semantics for these languages described in [1,5]. After that we discuss various applications of HiLog to object-oriented databases. Details are omitted in this abstract.

5 Proof Theory of HiLog

In this section, we present a resolution-based proof theory for HiLog. The following issues are examined: Skolem and Herbrand theorem, unification, and resolution. The discussion follows the development of resolution-based proof theory for predicate calculus [7]. Details can be found in the full paper.

6 Conclusion

We presented a logic, called HiLog, which combines in a clean declarative fashion the advantages of a higher-order language with the simplicity of first-order semantics. We have shown that HiLog can naturally support higher-order database queries, and that database query evaluators can be written in HiLog much more elegantly than in Prolog. We have also shown that HiLog provides a natural alternative semantics to such well-known database languages as COL and LDL, and can also be used as a platform for implementing some of the recently proposed object-oriented languages [9,18,17]. This, together with the extended resolution principle, makes HiLog a more natural platform for logic programming than the usual predicate calculus.

Acknowledgements: We are grateful to Thom Fruhwirth, Sanjay Manchanda and James Wu for their comments on the contents of this paper. Thanks also to the anonymous referee who suggested to investigate the relationship between HiLog and other database languages, such as COL and LDL.

References

[1] Abiteboul S. and Grumbach S. [1987] COL: A Logic-Based Language for Complex Objects, in *Proc. Workshop on Database Programming Languages,* Roscoff, France, September 1987, pp. 253-276.

[2] Abramson, H. [1988] Metarules and an Approach to Conjunction in Definite Clause Translation Grammars: Some Aspects of Grammatical Metaprogramming, in *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* R.A. Kowalski and K.A. Bowen, eds. Seattle, Washington, August 1988, pp. 233-248.

[3] Apt, K.R., Blair, H. and Walker, A. [1988] Towards a Theory of Declarative Knowledge, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 89-148.

[4] Apt, K.R. and Van Emden, M.H. [1982] Contributions to the Theory of Logic Programming, in *JACM* **29**, 841-862.

[5] Beeri C., Naqvi S., Shmueli O. and Tsur S. [1987] Sets and Negations in a Logic Database Language (LDL), MCC Technical Report, 1987.

[6] Chen, W., Kifer, M. , and Warren, D.S. [1989] HiLog: A First-Order Semantics of Higher-Order Programming Constructs, to appear in North American Conference on Logic Programming, Oct. 1989.

[7] Chang, C.L. and Lee, R.C.T. [1973] *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, New York.

[8] Chen, W. [1987] A Theory of Modules Based on Second-Order Logic, in *Proceedings of IEEE 1987 Symposium on Logic Programming,* San Francisco, September, pp. 24-33.

[9] Chen, W. and Warren, D.S. [1989] C-logic for Complex Objects, in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989,* pp. 369-378.

[10] Church, A. [1940] A Formulation of the Simple Theory of Types, *Journal of Symbolic Logic,* 5:56-68.

[11] Clark, K.L and McGabe, F.G. [1984] *MicroProlog: Programming in Logic,* Prentice Hall, 1984.

[12] Enderton, H.B. [1972] *A Mathematical Introduction to Logic,* Academic Press, Inc.

[13] Goldfarb, W.D. [1981] The Undecidability of the Second-Order Unification Problem, *Theoretical Computer Science.* 13:225-230.

[14] Henkin, L. [1950] Completeness in the Theory of Types, *Journal of Symbolic Logic,* 15:81-91.

[15] Fruhwirth, T. [1988] Type Inference by Program Transformation and Partial Evaluation, IEEE Intl. Conf. on Computer Languages, Miami Beach, FL, 1988, pp. 347-355.

[16] Fruhwirth, T. [1989] Polymorphic Type Checking in HiLog, manuscript in preparation.

[17] Kifer, M. and Lausen, G. [1989] F-Logic: A "Higher-Order" Logic for Reasoning about Objects, Inheritance, and Scheme, in: *1989 ACM SIGMOD Int. Conf. on Management of Data,* pp. 134-146.

[18] Kifer M. and Wu J., A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited), in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989,* 379-393.

[19] Krishnamurthy R. and Naqvi S., Towards a Real Horn Clause Language, *MCC Report ACA-ST-077-88,* Austin, TX, 1988.

[20] Kuper G., Logic Programming with Sets, in *Proc. 6th ACM Conf. on PODS,* San Diego, 1987, pp. 11-20.

[21] Kuper G., An Extension of LPS to Arbitrary Sets, *IBM Research Report,* 1987.

[22] Kuper G., Vardi M.Y., A New Approach to Database Logic, *PODS,* 1984.

[23] Lassez, J.-L., Maher, M.J. and Marriott, K. [1988] Unification Revisited, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 587-625.

[24] Lifschitz, V. [1988] On the Declarative Semantics of Logic Programs with Negation, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 177-192.

[25] Maier D., A Logic for Objects, in *Preprints of Workshop on Foundations of Deductive Database and Logic Programming,* ed. Jack Minker, Washington DC, August 1986.

[26] Maida, A. [1983] Knowing Intensional Individuals, and Reasoning about Knowing Intensional Individuals, in *Proc. 8th IJCAI,* West Germany, 1983, pp. 382-384.

[27] Maida, A. and Shapiro, S. [1982] Intensional Concepts in Propositional Semantic Networks, *Cognitive Science,* 6:4 (1982) pp. 291-330.

[28] Martelli A. and Montanari U., An Efficient Unification Algorithm, in *ACM Trans. on Progr. Lang. and Systems*, 4:2 (1982), pp. 258-282.

[29] Motro A., BAROQUE: A Browser for Relational Databases, *ACM Trans. on Office Information Systems,* 4:2 (1986), pp. 164-181.

[30] Miller, D.A. and Nadathur, G. [1986] Higher-Order Logic Programming, in *Proceedings of the Third International Conference on Logic Programming,* London, United Kingdom, July 1986, pp. 448-462.

[31] R. Montague, "The Proper Treatment of Quantification in English", in *Approaches to Natural Languages*, K.J.J. Hintikka, et al., (ed.), Dordrecht, 221-242, 1973.

[32] Maier, D., Warren, D.S. [1988] Computing with Logic, Benjamin/Cummings Publ., 1988.

[33] Nadathur, G. [1987] *A Higher-Order Logic as the Basis for Logic Programming,* Ph.D. Dissertation, University of Pennsylvania, June 1987.

[34] Przymusinski, T.C. [1988] On the Semantics of Stratified Deductive Databases, in *Foundations of Deductive Databases and Logic Programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 193-216.

[35] Van Emden, M.H. and Kowalski, R.A. [1976] The Semantics of Predicate Logic as a Programming Language, in *JACM* **23**, 733-742.

[36] Warren, D.H.D. [1982] Higher-Order Extensions to Prolog: Are They Needed? *Machine Intelligence,* 10:441-454.

[37] Warren, D.H.D. [1983] An Abstract Prolog Instruction Set, Report 309, AI Center, SRI International, Menlo Park, CA, October 1983.

How Stratification is Used in $\mathcal{LDL}$

Shamim A. Naqvi
MCC
3500 W. Balcones Center Drive
Austin, Texas 78759

Arpanet: shamim@mcc.com

Abstract

Recently, the author and his colleagues have been engaged in designing a logic programming language called $\mathcal{LDL}$ and providing declarative semantics for it. In the course of this work the idea of imposing a priority on non-monotonic operations and deriving a partitioning of the program universe has found surprising utility and has led to simple solutions to problems in ascribing declarative semantics to $\mathcal{LDL}$ programs. From this collective experience has emerged an understanding that this priority notion plays a role akin to assignment in logic query languages. This paper presents three instances of specific design and semantical issues, showing in each case how the priority notion leads to natural constructs in logic query languages, and affords straightforward solutions to problems in assigning declarative semantics to programs. However, stratification is not without its own problems. We critique stratification by exhibiting what may be termed as stratification anomalies.

The complete version of this paper appears in *Database Programming Languages*, Morgan-Kaufmann Publishers, 1989, and in Proc. of Workshop on Database Programming Languages, Salishan, Oregon, 1989.

1 Motivation

Recently, there has been a stream of results from the areas of logic programming, data and knowledge base systems, and non-monotonic reasoning in AI systems that are based on the idea of partitioning the universe of terms in such a manner as to *stratify* non-monotonic operations. (The use of stratification in constructing apparently consistent set theories, such as the von-Neumann-Bernays-Gödel, is well-known.) For example, this notion has been used to provide a sound, complete and efficient implementation of negation in data and knowledge base systems [CH85, Naq86, ABW88, VG86], collecting terms into sets in logic programs [BN*89], semantics of updates in deductive database languages [NK88], and higher-order variables in logic programming languages [KN88]. Some forms of McCarthy's circumscription schema have been shown to be equivalent to what is called stratified negation in this paper [Lif88]. Two logic programming languages for knowledge intensive applications under development, namely $\mathcal{LDL}$ [TZ86, NT89] and NAIL! [MUVG86], make extensive use of the stratification principle to allow natural constructs in language design and in providing declarative semantics for programs. (Not all the work in these areas uses stratification. For example, Manchanda [Man86, Man89, AV87] proposes similar update semantics, higher-order logic languages have been proposed by [M89, War89, Kif89, Nad87], and sets in logic languages have been studied by [AG87, Kup87, Kup88].)

From this collective experience and research results is emerging an understanding of stratification as a design principle for logical query languages. In this paper we provide an abstract view of the stratification principle and apply it to three technical problems that the author and his colleagues faced in designing $\mathcal{LDL}$ constructs and providing declarative semantics for $\mathcal{LDL}$ programs.

It is also shown that stratification is not with-

out its own limitations. In some cases, non-stratified programs exist for which no equivalent stratified programs exist, while in other cases, equivalent programs can be found but by adding exponentially many additional predicate symbols. The question remains open whether there exists a more universal design principle for logical query languages.

The contribution of the paper lies in providing a unifying view of an emerging body of knowledge. At the center of this new knowledge is the prime importance of the concept of stratification as a language design principle for Data and Knowledge Based programming systems. This principle is not without its problems. We critique stratification by exhibiting what may be termed as *stratification anomalies.*

2 Simple $\mathcal{LDL}$

$\mathcal{LDL}$ is a Horn clause language constructed from logical symbols $\forall, \leftarrow$ and conjunction represented by commas. The formulae (called *rules*) of the language have the form

$$A \leftarrow B_1, \ldots, B_n$$

where $A, B_1, \ldots, B_n$ are positive literals and all variables in the rule are universally quantified. The antecedent is usually referred to as the *body* and the consequent as the *head* of the rule. A rule with an empty body is called a *fact* and a rule with an empty head is called a *query*. A set of rules with head predicate symbol p are said to be a *definition* of the predicate (or literal) p.

A *program* is a finite collection of rules. The meaning of a program is the minimal model of the set of formulae comprising the program. Since a program is a Horn set it is immediate from Horn's theorem that a program has a unique minimal model. Thus simple $\mathcal{LDL}$ programs (of course this class is the same as pure Horn clause programs) have a unique declarative meaning.

Given a rule

$$A \leftarrow B_1, \ldots, B_n$$

and a *database* $\mathcal{DB}$, i.e., a collection of ground facts, we define the application of the rule to $\mathcal{DB}$ as the set

$$\{A\sigma \mid (\forall 1 \leq i \leq n)\ B_i\sigma \in \mathcal{DB}\}$$

where σ is a substitution of ground terms for variables. The set obtained by the ω-closure of rule application of all the rules $\mathcal{R}$ of P is called the *least fixpoint* of the program P and is denoted by $\mathcal{R}(\mathcal{DB})$. It is a theorem of van Emden and Kowalski [vEK76] that the least fixpoint of P is the minimal model of P.

It was recognized early that the language defined above will need to be embellished with additional constructs in order to render it applicable and usable in a wide setting. In particular, the need for introducing set terms as terms of the language, negated predicates in the body of a rule, imperative control constructs and higher-order procedures, was recognized. In each of these cases semantical issues arose which were resolved by using the idea of imposing a priority on the underlying operator. This is the subject of the next three sections.

3 Set terms

Our first extension of $\mathcal{LDL}$ involves introducing set terms into the universe of $\mathcal{LDL}$ programs. This extension, called *grouped sets*, allows sets to be constructed in the head of a rule and was first explored in [BN*89]. In standard mathematical notation we would denote a grouped set as $\{X \mid p(X)\}$ where $p(X)$ is some property of the elements X. The syntax for a grouped set term is $\langle t \rangle$ in which t is a non-ground first-order term. It is perhaps best to start with a particular example of a rule with a grouped set term. Consider the rule

$$p(Z, \langle X \rangle) \leftarrow b(Z, X).$$

Rules with $\langle\rangle$ in their heads are called *grouping rules*. A non-grouping rule is a rule without $\langle\rangle$ in its head. For a given extension of $b(Z, X)$ the above rule has the following meaning: Each distinct value of Z is in the first column and the set of X-values satisfying $b(Z, X)$ for that Z-value are grouped into a set term in the second column. Thus, the extension of p is as follows:

$$\begin{array}{ll} z_1 & \{x_{11}, x_{12}, \ldots, x_{1n_1}\} \\ z_2 & \{x_{21}, x_{22}, \ldots, x_{2n_2}\} \\ \ldots & \ldots \\ z_k & \{x_{k1}, x_{k2}, \ldots, x_{kn_k}\} \end{array}$$

where $\{b(z_i, x_{ij}) \mid 0 \leq i \leq k; 0 \leq j \leq n_i\}$ is the extension of the relation (i.e., predicate b).

Generally, consider a rule of the form

$$p(t_1, \ldots, t_n, \langle Y \rangle) \leftarrow body(\overline{X}, Y)$$

where $\overline{Z}$ are all the variables within $t_1, \ldots, t_n$ and $\overline{X}$ are the variables appearing in the body except for

Y; Z may, however, include Y. Let $\overline{V}$ be the set of all the variables in the rule. The meaning of the rule is given as follows. Construct a relation, R, by evaluating the body. Next partition R horizontally for each distinct combination of values in $\overline{Z}$. Then group all the Y-values in each partition into a set. The derived relation p has a tuple for each distinct partition of the relation R; the first column of p has a distinct combination of values of $t_1, \ldots, t_n$ and the second column has the corresponding grouped set.

An unrestricted use of grouping may yield programs that do not have any meaning in our semantics. Consider, for example, the following program:

$$\mathtt{p}(\langle \mathtt{X} \rangle) \leftarrow \quad \mathtt{p}(\mathtt{X}).$$
$$\mathtt{p}(\mathtt{a}).$$

Proposition[BN*89]: The above program does not have a model in the given $\mathcal{LDL}$ universe. ■

What this means is that we have to rule out such programs. The device we use towards this purpose is the priority principle stated as

> In grouping rules we require that all the relations in the body of a grouping rule be computed before the relation in the head of the rule is computed.

In $\mathcal{LDL}$ this is called the *stratification condition* for reasons that will become obvious momentarily.

Let us define a partial order on the predicates of a program as follows.

$p > q$ if there exists a rule of the form

$$p(\ldots, \langle \ldots \rangle) \leftarrow \ldots, q(\ldots), \ldots$$

$p \geq q$ for non-grouping rules of the form

$$p(\ldots) \leftarrow \ldots, q(\ldots), \ldots$$

Program P is *admissible* if there is no sequence of predicate symbols in the rules of P of the form

$$p_1 \theta_1 p_2 \ldots \theta_{k-1} p_k \theta_k p_1$$

where $\forall i (1 \leq i \leq k) \theta_i \in \{<, \leq\}$ such that $\exists j (1 \leq j \leq k) \theta_j$ is $<$.

A partitioning $\{L_i\}_{i \geq 0}^{n}$ of the predicate symbols of a program P is called a *layering* if for all $p, q \in P$

1. if $p \geq q, p \in L_i, q \in L_j$ then $i \geq j$, and
2. if $p > q, p \in L_i, q \in L_j$ then $i > j$.

Observe that there may be more than one layering for a given program.

Theorem [BN*89]: A program P is admissible if and only if it is layered. ■

We now define a notion of standard model, $\mathcal{M}_P$ of an admissible program P with layering $\mathcal{L}_1, \ldots, \mathcal{L}_n$ on a database $\mathcal{DB}$. Note that a layering of a program produces a partitioning of the rules of the program. Let $\mathcal{L}_i(\mathcal{DB})$ denote the application of the rules in the program partition induced by $\mathcal{L}_i$ on the database $\mathcal{DB}$.

$$\mathcal{M}_1 = \mathcal{L}_1(\mathcal{DB})$$
$$\mathcal{M}_2 = \mathcal{L}_2(\mathcal{M}_1)$$
$$\vdots$$
$$\mathcal{M}_n = \mathcal{L}_n(\mathcal{M}_{n-1})$$

Using this notion of a standard model it was possible for Beeri, Naqvi, Shmueli and Tsur [BN*89] to prove that admissible programs have a standard model, that the above construction yields a standard model and that a standard model is a minimal model of the program. They also advance reasons to consider the standard model as the canonical meaning of the program.

Shmueli and Naqvi [SN87] consider the power of the grouping operator under the stratification condition. Consider the program P defined by

$$\mathtt{p}(3, \{1\}).$$
$$\mathtt{p}(\mathtt{G}, \langle \mathtt{Y} \rangle) \leftarrow$$
$$\mathtt{p}(\mathtt{V}, \mathtt{G}), \mathtt{member}(\mathtt{Z}, \mathtt{G}), \mathtt{member}(\mathtt{Y}, \{\mathtt{f}(\mathtt{Z}), \mathtt{g}(\mathtt{Z})\}).$$

Note that this program is non-stratified with respect to the grouping operator. The minimal model of this program is shown in [SN87].

Theorem[SN87]: There does not exist a stratified program which has the model $\mathcal{M}$ of P. ■

Shmueli and Naqvi further show that in this particular case we may, however, add additional predicate symbols and obtain a program, P', whose minimal model $\mathcal{M}'$, when restricted to the predicate symbols of P, is equivalent to $\mathcal{M}$ (e.g., if we add the

predicate symbol q to P, we will delete all facts with the predicate symbol q from $\mathcal{M}'$). They leave open the question whether, in general, by adding predicate symbols we can always find a stratified program having the same unique minimal model as a given non-stratified program.

4 Negation

Horn clause programs with negation are strictly more powerful than programs without negation [Imm86, Var82]. We may therefore consider allowing the body of a rule to contain negated literals and ask what declarative semantics ensue from such an extension. Immediately we have a problem in that the uniqueness property of the minimal model is lost, e.g., the program

$$a \leftarrow \neg b$$

has two minimal models $\{a\}$ and $\{b\}$ and their intersection is not a model. If we are to adhere strictly to the maxim that the meaning of a program is given by a unique model then we must fix upon one of these two models and discard the other. Once again we may take lesson from the previous section and define a notion of a standard model. People have commented that the model $\{a\}$ is more "natural" than the model $\{b\}$. The argument is based upon the observation that since the programmer chose to represent the disjunction $a \vee b$ as $a \leftarrow \neg b$ and not as $b \leftarrow \neg a$, this choice of representation is a reflection of the programmers *intended* model. The question is how we can formalize this notion of the user's intended model. This affords us another use of the stratification rule which, in this context, is formulated as follows: The definition of a literal must appear prior to its negation. Let us agree to refer to such uses of negation as *stratified negation.*

We can make the idea of stratified negation more precise by defining relations $\geq, >$ on the predicate symbols of a program P as follows.

$p \geq q$ if there exists a rule of the form

$$\mathrm{p} \leftarrow \ldots, \mathrm{q}, \ldots$$

$p > q$ if there is a rule of the form

$$\mathrm{p} \leftarrow \ldots, \neg \mathrm{q}, \ldots$$

Program P is *admissible* if there is no sequence of predicate symbols in the rules of P of the form

$$p_1 \theta_1 p_2 \ldots \theta_{k-1} p_k \theta_k p_1$$

where $\forall i (1 \leq i \leq k) \theta_i \in \{\geq, >\}$ and some θ_j is $>$, $(1 \leq j \leq k)$.

As in section 3 we can show that the admissibility condition on a program P imposes a partitioning of the predicate symbols of P.

We are interested in declaratively characterizing the models for a given program with negation. Assume that we have an admissible program P with layers $L_0, L_1, \ldots, L_n$. Without loss of generality, we may assume that L_0 is free of negation, i.e., has no negated predicates in any rules. It is possible to make this assumption because, trivially, we may start with the empty layer for L_0 for any given program. Assume that P has two layers L_0, and L_1. Thus, L_0 must have a unique least model, M_0. For every rule in L_1 of the form

$$A \leftarrow B_1, \ldots, B_n$$

where the body may contain negated predicates, make A true over every element of the universe of the program. We can do this because A occurs only positively in L_1. Let $\mathcal{A}$ denote this set, i.e., set of positive atoms $A(\ldots)$ constructed from all the terms of the universe of the program. Define

$$M_1 = M_0 \cup \mathcal{A}$$

Clearly, M_1 is a model of $L_0 \cup L_1$.

We shall now define certain submodels of M_1. These will be called *standard models.*

$$M_{11} = M_0 \cup N_1 \text{ where } N_1 \subseteq \mathcal{A}$$

$$M_{12} = M_0 \cup N_2 \text{ where } N_2 \subseteq \mathcal{A}$$

$$\vdots$$

Thus M_{11}, M_{12}, etc., differ from M_1 in the extension of the predicate A, i.e., we consider subsets of $\mathcal{A}$ such that M_{11}, M_{12}, etc., are models of $L_0 \cup L_1$.

Lemma [NT89]: Let M_{11} and M_{12} be two standard models for a program. Then $M_{11} \cap M_{12}$ is a standard model for that program. ■

Theorem[NT89]: A layered program has a least standard model. ■

The next question one may ask is the following: Given a non-stratified program having a unique minimal model does there exist a program with stratified negation having the same unique minimal model? The answer to this question is given by Imielinski and Naqvi [IN88]. They show that, if no additional predicate symbols are allowed to be added to a program, then there exists a program having a unique minimal model but for which no equivalent stratified program exists, i.e., a program having the same unique minimal model. However, if we do allow additional predicate symbols to be added, then a program can always be found whose unique minimal model, when restricted to the predicate symbols of the original program, is equivalent to the unique minimal model of the original program. The number of additional predicate symbols that have to be added is bounded exponentially in the size of the program.

5 Imperative Predicates

Our third extension of $\mathcal{LDL}$ is to define certain imperative predicates, called *actions*, to allow database updates and conventional control constructs such as the conditional *if-then* and the iterative constructs. Semantics of these constructs are provided in [NK88] by resorting to a Dynamic Logic [Har79]. Let us define informally the imperative actions in $\mathcal{LDL}$.

1. If α is a positive ground atomic formula then $+\alpha$ and $-\alpha$ are actions.
2. If α and β are actions and P is a predicate then if(P then α), $(\alpha;\beta)$, and forever(α) are actions.

Actions have the following intuitive meanings:

$+p(t_1,\ldots,t_n)$ inserts the n-tuple $\langle t_1,\ldots,t_n\rangle$ into the n-ary base relation p.
Similarly, $-p(t_1,\ldots,t_n)$ deletes the indicated tuple from the specified relation.

if(P then α) means *do α if P is true, otherwise do nothing.*

$(\alpha;\beta)$ means *do α followed by β.*

forever(α) means repeat α an unbounded number of times.

We refer the reader to Naqvi *et al.* [NK88] where, using Dynamic Logic, declarative semantics are provided for $\mathcal{LDL}$ with these imperative constructs.

We now have an informal understanding of the imperative predicates of $\mathcal{LDL}$. Unfortunately, we can show that unrestricted use of imperative constructs leads to programs without models. We start our exposition by considering, by way of example, a program with a rule of the form

$$a \leftarrow a;\alpha.$$

Note that this rule requires that a be computed before α but where the computation of α depends upon the extension of a. This is an apparent inconsistency. Another example is provided by the rule

$$a \leftarrow \text{forever}(\ldots,a,\ldots)$$

in which, once again, the computation of a in the body of the rule depends upon the extension of a in the head predicate. Clearly, we need to exclude programs in which imperative predicates refer to predicates that also appear as derived relations, i.e., we need to make the computation of the predicates in the body independent of the predicate in the head of that rule. By now, we recognize, from the above two examples, that the computation of predicates must be *stratified*. As before, we shall define an ordering on the predicates of the program to enforce stratification with respect to imperative predicates and define the class of legal programs.

We define an ordering between predicate symbols as follows.

- If there is a rule of the form

  $$A \leftarrow \alpha$$

  where $\alpha = \ldots,B,\ldots$ and α is a predicate then A precedes B written as $A \leq B$. If α is an action then then A is said to strictly precede B, and written as $A < B$.

- If there is a rule of the form

  $$head \leftarrow A_1,\ldots,A_n;B_1,\ldots,B_k$$

  then each $A_i, i = 1,\ldots,n$ strictly precedes all $B_j, (\forall 1 \leq j \leq k)$, i.e., $(A_i < B_j)$; otherwise each $(A_i \leq B_j)$.

A program is *legal* if there does not exist an ordering of its predicate symbols

$$p_1\theta_1 p_2\theta_2 \ldots p_k\theta_k p_1$$

where $\forall i(1 \leq i \leq k)\theta_i \in \{<,\leq\}$ such that $\exists j(1 \leq j \leq k)\theta_j$ is $<$.

As before, stratification implies a partitioning on the rules of a program as follows. Using the notion of legal program, it was possible for Naqvi *et al.* [NK88] to provide a definition of a minimal model of the program.

We now show how non-stratified uses of set grouping and negation can be simulated by using update and imperative actions. This shows some of the limitations in the power of stratification as a universal design principle. In the last two sections we have considered the power of stratified grouping and negation. We have seen that in the case of set grouping, we have a program which has a unique minimal model but which does not have an equivalent admissible program. In many cases, however, we can use imperative predicates to express an inadmissible collection of grouping rules as an admissible program. We shall see an example of this below. In the case of negation we saw that every non-stratified program with a unique minimal model can be expressed, possibly by using additional predicate symbols, as an admissible program [IN88]. However, we may need exponentially many (in the size of the predicate symbols in the program) additional predicate symbols. We shall show below an example of an inadmissible program with negation which can be easily re-expressed as an admissible program without having to add many extra predicate symbols.

As our first example, consider the program to define a *loyal republican* to be a person all of whose ancestors were loyal republicans. Furthermore, let it be given that Abraham Lincoln was a loyal republican. We assume the `ancestor` relation to be given from which we can easily derive a new relation `ancestors(X,Y)` giving the set of ancestors `Y` for the person `X`. We may express the "loyal republican" problem by using universal quantification as follows:

$(\forall X)$`loyalRepublican(X)` $\leftarrow$
    $(\forall W \in S)$(`ancestors(X,S)`,
        `loyalRepublican(W)`.

`loyalRepublican(lincoln)`.

where S denotes the set of all ancestors of X.

The problem is how to simulate the universal quantifier, $\forall$ in the body of the first formula, in a Horn clause language? An inadmissible approach is shown in the example below where the set grouping is performed within a recursive set of rules.

Example: Inadmissible program for simulating universal quantification in rules.

```
% Group all ancestors for each X.
ancestors(X, ⟨Y⟩) ← ancestor(X, Y).

% For each X, group all ancestors that are
loyal republicans.
a(X, S, ⟨W⟩) ←
    ancestors(X, S), loyalRepublican(W),
        member(W, S).

% If the set, grouped in predicate a is
equal to S then X is a loyal republican.
loyalRepublican(X) ← a(X, S, S1), S = S1.
```

■

The correct solution to the loyal republican problem is shown below.

```
ancestors(X, ⟨Y⟩) ← ancestor(X, Y).

loyalRepublican(lincoln).

aux(⟨X⟩) ← ancestors(X, S), ¬bad(S).

bad(S) ←
    member(W, S), ¬loyalRepublican(W).

?forever(if(aux(X), +loyalRepublican(X)
        then true)).
```

Note that in this solution we use the formal equivalence of

$$\neg(\forall X)p(X) \equiv (\exists X)\neg p(X)$$

■

For the second example consider the problem of identifying the set of *solvable* nodes, in a directed graph. A solvable node is a node in the graph that is not a member of any infinite path, i.e., the node is neither a member of a cycle nor follows a member of a cycle. To write this in Datalog with negation [Ull88] we could define two predicates: solvable nodes *s(Y)* and unsolvable nodes *us(Z)*. A node is solvable if it follows a solvable node and it is itself not unsolvable; similarly, a node is unsolvable if it follows a node that is not solvable. Below we show the program, G, to compute the solvable nodes in a directed graph.

```
s(Y) ← s(X), g(X, Y), ¬us(Y).
us(Z) ← g(W, Z), ¬s(W).
```

Note that this program has a unique minimal model for any given database, i.e., any given directed graph. Further this program is non-stratified. The following is the equivalent $\mathcal{LDL}$ program, P'':

```
s(Y) ← s(X), g(X,Y), ¬us'(Y).
us(Z) ← g(W,Z), ¬s'(W).
?forever(
  if (s(Y), us(Z), s'(Y1), us'(Z1))
  then(−s'(Y1), +s'(Y),
    −us'(Z1), +us'(Z))).
```

Note that `s'` and `us'` are base predicates in the above program with an empty set of tuples initially.

We claim that the constructed model $M_{P''}$ of the program P'' is such that when it is restricted to the predicate symbols of program G it is equal to the constructed model M of G.

6 Conclusion

The use and power of stratification in the design and semantics of languages without assignments has been shown in this paper. Of course destructive assignment completely solves these problems but has only operational semantics. Single assignment, while affording declarative semantics, is not by itself powerful enough to solve these problems. Thus stratification may prove to be a useful middle ground between single and destructive assignment, in that it partially solves certain design problems and admits declarative semantics.

Acknowledgements

I thank Hassan Aït-Kaci for motivating me to write this report, and to my co-workers on the $\mathcal{LDL}$ project in the Languages Group at MCC for making $\mathcal{LDL}$ possible. Thanks also to Mike Carey for suggesting several improvements of presentation.

References

[AG87] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Proc. of a workshop on Database Programming Languages*, Roscoff, 1987.

[AV87] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. of 6th ACM PODS*, San Diego, 1987.

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, Morgan-kaufman, 1988.

[BN*89] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. Submitted for publication to the Journal of Logic Programming, 1989.

[Can15] G. Cantor. *Contributions to the Theory of Transfinite Numbers.* Open Court, Chicago and London, 1915. A Translation of Cantor 1895–7 by P.E.B. Jourdain.

[CH85] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[Har79] D. Harel. *First-Order Dynamic Logic.* Springer-Verlag LNCS 68, 1979.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68, 1986.

[IN88] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proceedings of the 7th Annual Association of Computing Machinery Symposium on Principles of Database Systems*, 1988.

[Kif89] M. Kifer and G. Lausen. F-Logic: A higher-order logic for reasoning about objects, inheritance, and scheme. In *ACM SIGMOD*, Portland, 1989.

[KN88] R. Krishnamurthy and S. Naqvi. Nondeterministic choice in datalog. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988.

[Kup87] G. Kuper. Logic programming with sets. In *Proc. of ACM PODS*, San Diego, 1987.

[Kup88] G. Kuper. An extension of LPS to arbitrary sets. IBM Technical Report, 1988.

[Knu69] D. Knuth. *The Art of Computer Programming.* Volume 1 : Fundamental Algorithms, Addison-Wesley, 1969.

[Lif88] V. Lifschitz. On the declarative semantics of logic programs with negation. In *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufman, 1988.

[Man86] S. Manchanda and D. Warren. A logic-based language for database updates. In *Foundations of Deductive Databases and Logic Programming*, (J. Minker, editor), Morgan-Kaufmann, 1987.

[Man89] S. Manchanda. Declarative expression of deductive database updates. In *Proc. of 8th ACM PODS*, Philadelphia, 1989.

[M89] S. Manchanda. Higher-order logic as a data model. In *Workshop on Database Programming Languages*, Oregon, 1989.

[MUVG86] K. Morris, J. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings 3rd Int Conference on Logic Programming*, pages 554–568, Springer-Verlag LNCS 225, 1986.

[Nad87] G. Nadathur. A higher-order logic as a basis for logic programming. Ph. D. dissertation, Univ. of Pennsylvania, 1987.

[Naq86] S. Naqvi. A logic for negation in database systems. In *Proceedings Workshop on Logic Databases*, 1986.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proceedings 7th Association of Computing Machinery Principles of Database Systems*, 1988.

[NT89] S. Naqvi and S. Tsur. A Logical Query Language for Data and Knowledge Bases. *W. H. Freeman and Co., New York*, 1989.

[SN87] O. Shmueli and S. Naqvi. Set grouping and layering in horn clause programs. In *Proceedings of the 4th International Conference on Logic Programming*, 1987.

[TZ86] S. Tsur and C. Zaniolo. LDL: a logic-based data-language. In *Proceedings 12th Conference on Very Large Data Bases*, 1986.

[Ull88] J. Ullman. *Principles of Data and Knowledge-Base Systems.* Volume 1, Computer Science Press, 1988.

[Var82] M. Vardi. The complexity of relational query languages. In *Proceedings of the 14th Symposium on Theory of Computing*, pages 137–146, 1982.

[vEK76] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association of Computing Machinery*, 23(4):733–742, 1976.

[VG86] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *Proceedings 3rd IEEE Symposium on Logic Programming*, pages 127–139, 1986.

[War89] D. Warren. HiLog as a Platform for Database Languages. In *Workshop on Database Programming Languages II*, Oregon, 1989.

Database Programming by Formal Refinement of Conceptual Designs[1]

Joachim W. Schmidt, Ingrid Wetzel
Fachbereich Informatik
Universität Frankfurt
D-6000 Frankfurt a.M.
F.R. Germany
schmidt@dbis.informatik.uni-frankfurt.dbp.de

Alexander Borgida
Dept. Comp. Science
Rutgers University
New Brunswick, NJ08904
USA
borgida@aramis.rutgers.edu

John Mylopoulos
Dept. Comp. Science
University of Toronto
Toronto M5S 1A4, Ont.
Canada
jm%utai%toronto.csnet@relay.cs.net

Abstract

Developing an Information System involves requirements gathering, design specification, and, last but not least, the implementation and maintenance of a large database application program. In the DAIDA project, the domain and requirements are modeled using the temporal language Telos, the conceptual design is specified using TDL, and the target implementation language is DBPL; DBPL is a procedural database programming language with persistent values of a wide variety of types, with database transactions as first-class language constructs, and emphasizing database and program modularity. After a brief description of TDL and DBPL, this paper concentrates on the iterative refinement of designs into programs, based on J.-R. Abrial's formal framework of Abstract Machines and Generalized Substitutions (for an extended discussion of our approach, see [10]).

1 Motivation and Setting

The authors and their collaborators are involved in project DAIDA, whose goal is to build a novel software engineering environment for developing and maintaining Information Systems [7]. It is generally agreed that the development of software systems, including Information Systems, involves stages such as requirements definition, design and implementation/testing. One of the key features of the DAIDA project is the use of three specific (kinds of) languages for the description of the software at each stage: 1) a knowledge representation language for domain analysis; 2) a semantic data model for the conceptual design of states and transitions; and 3) an imperative programming language for efficient management of large, shared data sets. Among the expected contributions of the project will be a better understanding of the problems and tools/techniques for moving from a problem specification to the implementation of an Information System with data management components of significant complexity but relatively straight-forward procedural requirements.

For determining the user's requirements, we have argued that it is important to arrive at, and capture, an understanding of the application domain [8]. For this purpose, we have progressed through a series of languages, the latest called Telos, which allows one to represent a temporal model of the application domain in an assertional knowledge representation language [11]. Telos uses the paradigm of objects related by temporally qualified relationships to describe the entities and activities (both generic and specific) that occur in a particular world. The crucial fact about Telos is that *every aspect* of the description is associated with *time intervals*: the occurrence of activities, the membership of objects in classes, the relationships between objects, activities, etc.; in fact, in a recursive process, not only is the domain viewed temporally, but also the description itself is temporally indexed.

There are several problems with jumping from the user's conceptual requirements directly to the code of a database program :

- not everything seen by the user in the world needs to be part of the final software system - much of it is contextual information;

[1]This is a preliminary report on results from DAIDA, a project supported in part by the European Commission under ESPRIT contract #892.

- Telos is meant to capture the ontology of the user domain (so a project is viewed as an activity) while a database usually captures information *about* the domain (so there may be a *data* class corresponding to some activities in the world);
- in most environments there are multiple needs/goals that are being achieved, and these need to be integrated;
- finally, the descriptions in Telos and DBPL are just too widely different in style.

TDL therefore is concerned with the intermediate stage of *conceptual design*: identifying, specifying, and relating the data and procedural *components* of an information system, which itself is to be implemented in DBPL.

In this paper we shall briefly discuss some of the features of the TDL [13] and DBPL [21] languages, and then outline some of the innovative tools and techniques that are used to iteratively map TDL designs into DBPL programs by successive formal refinements within the framework of Abrial's Abstract Machines and Generalized Substitutions [3], [1].

2 TDL: A Language for Conceptual Designs

TDL is a *program specification language* which can be thought of as an *assertional* version of the Taxis language [18], extended with set-valued expressions, whose form was influenced by DBPL and Abrial's Abstract Machines [2], [26].

For data design, a number of different kinds of data classes can be defined in TDL. In addition to the usual *base classes* and *enumerated classes*, TDL distinguishes

- *aggregate classes*, which are essentially labeled Cartesian products, and whose instances have equality decided structurally; and
- *entity classes*, which have an associated extent: a set of objects with intrinsic, unchanging identity. The definition of entity classes specifies, among others, the attributes applicable to their instances, as well as their ranges.

Moreover, TDL has a small set of built-in *attribute qualifiers*, which allow constraints to be placed on attribute values, and their evolution over time:

- UNCHANGING attributes cannot be modified, once assigned a value;
- CHANGING attributes, in contrast, may get different values as states change;
- UNIQUE attributes are required to have different values for all instances of the class (i.e., can act as "keys" in relational terms).

These qualifiers correspond to constraints that are frequently useful in the practice of data base system design. (Note also the contrast with Telos, where a user of the language may create arbitrary new attribute qualifiers in order to abbreviate constraints – this, because attributes are first class citizens in Telos.)

For *state change and evaluation* , TDL provides for the definition of two kinds of procedures:

- em functions, which are evaluated with respect to a single database state, and leave it unchanged while returning a value; and
- *transactions*, which specify atomic transitions between states free of inconsistencies.

The parameters of transactions appear as attributes with the property categories IN and OUT indicating whether the parameter is used to communicate information into or out of the transaction. A specification is intended to associate with every transaction a set of acceptable state transitions. One standard approach is that of asserting constraints on the values of the state variables (attribute functions, parameters and class extents). For transactions, this is the familiar precondition/postcondition notation relating the initial and final states; this technique is widely used in such specification languages as VDM [16], Z [23], [15] and Larch [14]. In TDL, these assertions are expressed in the usual Taxis-like notation, under two attribute categories:

- GIVEN: conditions required to hold in the initial state when the transaction is invoked;
- GOALS: conditions required to hold in the final state.

Both conditions are logical assertions, with goals able to relate values in both the initial and final state. For this purpose, we use the standard technique of distinguishing initial and final state values of variables by marking with the suffix ' the values to be evaluated in the final state.

The insertion and removal of objects from the extents of classes can be expressed, among others, through two attribute categories, PRODUCES and CONSUMES, that assert that the attribute value is (is not, resp.) an instance of the class specifying its range. An example of a TDL specification is given in the Appendix, Fig. 1. Additional conditions for "normal-case first" abstraction / exception handling are discussed in [6].

Transactions describe atomic state transitions. Especially in business and administrative environments, it is often necessary to present more global constraints on sequences of actions (e.g., an employee needs to have a physical examination before completing the pay form). In order to support such sequencing specifications, we use *scripts*, which are composed of *locations/states*, and *transitions* connecting the states, where each transition contains an atomic transaction. More detailed discussions of scripts and communication facilities between them can be found in [13], [5], [12], [19].

It should come as no surprise that subclass hierarchies are supported and their use is encouraged throughout the design, according to the methodology described in [9].

3 DBPL: A Type-Complete Database Programming Language

At the other end – implementation – we have available the database programming language DBPL [21], a successor of Pascal/R [20]. DBPL integrates a set- and predicate-oriented view of database modeling into the system programming language Modula-2 [25]. Based on integrated programming language and database technology, it offers a uniform framework for the efficiency-oriented implementation of data-intensive applications.

DBPL is a strongly and statically typed language equipped with a rich set of base types (Integer, Real, Boolean, Character, ...) which can be combined by a variety of type constructors (array, record, variants, set/relation). DBPL is *type-complete* in the sense that its type constructors can be applied to any base type and can be combined freely (e.g., to define nested sets). Furthermore, any type can be used in any context, e.g., in local scopes or in parameter positions.

A central design issue of DBPL was the development of abstraction mechanisms for database application programming [22], [17]. DBPL's bulk data types are based on the notion of (nested) sets, and first-order predicates are provided for set evaluation and program control. Particular emphasis had been put on the interaction between these extensions and the type system of Modula-2. DBPL extends Modula-2 *orthogonally* in three dimensions:

- bulk data management through a data type *set/relation*;
- abstraction from bulk iteration through *first order* predicates , iterators and set expressions;
- *database modules* for sharing and persistence of DBPL objects and *transactions* for failure recovery and concurrency control.

DBPL programs can be structured into sets of modules with well-controlled interfaces through which DBPL objects are exported and imported. Since data-intensive applications require the persistence and sharing of states across program executions, DBPL has the notion of *database module*, which makes all its variables persistent. Persistent variables are protected by disallowing access from outside a transaction. A first impression of the overall software engineering quality of DBPL implementations is given by the complete DBPL example presented in the Appendix, Fig. 2.

4 From TDL Designs to DBPL Implementations

A major effort of the DAIDA project concentrates on the production of high-quality database application software. DAIDA makes the following assumptions about this software production process:

- for a given TDL-design there may be many substantially different DBPL implementations, and it needs human interaction to make and justify decisions that lead to efficient implementations;
- the decisions are too complex to be made all at once – let alone automatically; it needs a series of refinement steps referring to both data and procedural objects;
- the objects and decisions relevant for the refinement process need to have formally assured properties;
- these properties should be recorded to support overall product consistency and evolution.

To meet the above requirements, DAIDA relies heavily on current work of J.-R. Abrial [3], [1]: TDL designs are translated into Abstract Machines with states represented by mathematical objects like functions and sets, and state transitions defined by Generalized Substitutions; then the formal properties of Machines and Substitutions are assured and organized by Abrial's interactive proof assistant, the B-Tool [4].

In the remainder of the paper we report on our first experience with a rule-based Mapping Assistant that helps in refining TDL designs into DBPL implementations via Abstract Machines. First we outline the formal framework and then we discuss in some detail the refinement process itself.

For an example we use part of the TDL-design given in the Appendix, Fig.1. ResearchCompanies consists of three entity class definitions (Companies, Employees, Projects) and one transaction definition (HireEmployee). One of the invariants asserts that the projects a company is engaged in are exactly those which have this company as a consortium member. Another invariant enforces the projects on which an employee works to be a subset of the projects his company is engaged in. For some of the classes UNIQUE attributes are specified, while for the class employees there is no such attribute.

4.1 Abstract Machines and Generalized Substitutions

Abrial's framework for specifying and developing software systems is based on the notion of Abstract Machines, a concept for organizing large specifications into independent parts with well-defined interfaces [3]. An Abstract Machine is a general model for software systems and is defined by a state (the statics of the system) and a set of operations modifying the state (the dynamics).

The *statics of a system* include the definition of three main Abstract Machine components: Basic Sets, Variables, and system Invariants. *Basic Sets* define time-invariant repositories of objects (types) that may occur in our application. In a first version of a refinement, they may be completely abstract, defined only by their name. The *Context* allows us to freeze the definition of basic sets as soon as we make decisions on their properties. *Variables* are either time-dependent subsets of the Basic Sets and model the state of an Abstract Machine, or they are binary relations that associate elements of Basic Sets (viewed as functions from the domain to the range). Finally, the *Invariants* express the static relationships within the system. They are defined in terms of predicates and sets (with a rich choice of operators for set, relation, and function (re-) definition).

The *dynamics of a system* are defined by the Operations of an Abstract Machine. The definition of an operation, for example, HireEmployee, reads as follows[2]

OPERATIONS HireEmployee (name, belongs, works) =
PRE $name \in EmpNames \wedge belongs \in companies \wedge works \in \wp\ (engagedIn(belongs))$
THEN ANY e IN (Employees - employees)
THEN (empName(e), worksOn(e)), belongsTo(e) $\mid\!\leftarrow$ *(name, works, belongs)* $\|$
$employees \mid\!\leftarrow employees \cup \{e\} \;\|$ *HireEmployee* $\mid\!\leftarrow e$
END
END HireEmployee;

Operations are specified by the constructs of the Generalized Substitution Calculus with semantics defined by the weakest precondition under which a construct fulfills a given postcondition. In our example operation, the ANY-substitution expresses an unbounded non-deterministic substitution. It chooses an arbitrary fresh member from (Employees – employees), i.e., from the set of non-instantiated elements considered for employee representation. Next, the (attribute-) functions for that new employee element are

[2]Notation: $\wp$ indicates "power set", $\|$ is parallel composition, and $\mid\!\leftarrow$ is parallel assignment/substitution.

redefined in parallel. The "type conditions" on the input parameters are expressed by the preconditions of the Generalized Substitution.

4.2 Consistency of Designs

With Abstract Machines we are in a position to prove the consistency of specifications, a task that is of particular importance for heavily constrained database applications [24]. In general, we have to prove for each operation that it preserves the invariants of the Machine. For example, during such a proof of the transaction HireEmployee, the *Employees* class invariant *onEmpProj* gives rise to the following lemma: *belongs* $\in$ *companies* $\wedge$ *works* $\in$ $\wp$ *(projects)* $\Rightarrow$ *works* $\in$ $\wp$ *(engagedIn (belongs))*. If such a lemma were not provable (e.g., because the designer had mistakenly omitted the needed precondition) we would be alerted to the error and could correct it at this stage.

4.3 On Data and Procedure Refinement

In our setting, refinement is defined between Abstract Machines having the same signature, i.e., the same number of operations and parameters. A Machine, A (the more abstract one) is said to be refined by a Machine C (the more concrete one) iff the refinement predicate, P_{AC}, that relates the state variables of both machines is proven to be an invariant under all operations, S_A and S_C, of both machines, A and C. The proof obligation also considers the case of a non-deterministic operation in machine A and a less non-deterministic one in machine C. A mapping process usually consists of a series of procedure- and data-oriented refinement steps from one Abstract Machine into another. The refinement process can be directed interactively by the user and controlled formally by the proof assistant.

Data refinement is defined in two steps:

- a data representation has to be chosen that is "closer" to the concrete data structures being offered by the target programming language used to implement the specification;
- the relationship between the more concrete state space and the more abstract one has to be expressed by a refinement predicate that involves variables of both machines.

Usually, changes in data representations imply refinements of operations. An operation S_A is refined by an operation S_C "if S_C can be used instead of S_A without the 'user' noticing it" [3]. More formally, this means that all postconditions R that are established by S_A are also established by S_C. Within the framework of Generalized Substitutions, algorithmic refinement is formally defined as a partial order relation between substitutions [3].

Currently, we are experimenting with a standard refinement strategy consisting basically of three steps [10]. In each step we decide upon one major task in database modeling:

1. identification of data objects in extents of varying cardinalities;
2. alternative ways of organizing data using the data type structures of the language; and
3. introduction of those constraints for variables, parameters, etc. that can be checked efficiently and at appropriate times (static typing).

After these three refinement steps we are in a position to automatically transform the final Abstract Machine into an equivalent DBPL program.

For the first refinement step of our example, we make the rather specific but common decision to utilize the uniqueness constraints on properties for companies and projects for data identification. For employees, where the application does not provide such a constraint, our refinement introduces an additional property, *empId*, for which the implementation assures uniqueness. Due to the change in the data representation, the operation HireEmployee becomes less non-deterministic: the arbitrary ANY-substitution is replaced by some operation, newEmpId, that provides a fresh EmpId value, which is then associated with the required properties through function extension. Our refined data and procedure representation can be proven to meet all the invariants laid down in the initial Abstract Machine.

Having decided upon identification, we are now ready for the second refinement step: designing the central data structures of our implementation. While our first Abstract Machine has a purely functional view of data, we now refine into a representation that is based on Cartesian products. It is at this level that we have to decide whether our data structures have set-valued attributes (i.e., will finally be mapped into nested DBPL relations) or will be flattened. In our example we opt for the first alternative.

On our way to implementation there is a third refinement step left that deals with data typing. Since DBPL is a strongly and statically typed language we want to refine the variables to become partial functions over the Basic Sets instead of total functions that range over other variables with time-varying cardinalities. For this reason, the precondition of the hireEmployee operation is weakened to constraints that will finally become the parameter types of the HireEmployee transaction. However, to meet the inherited specification we have to strengthen our constraints by introducing a *conditional substitution* which, due to the semantics of Generalized Substitutions, will finally result in a first-order query predicate on the variables that represent database states and transaction parameters (compare DBPL transaction, HireEmployee, Appendix, Fig. 2):

OPERATIONS HireEmployee (name, belongs, works) =
PRE name $\in$ *EmpNames* $\wedge$ *belongs* $\in$ *CompNames* $\wedge$ *works* $\in$ *ProjIdRelType*
THEN IF belongs $\in$ *companies* $\wedge$ *works* $\in$ $\wp$ *(engagedIn(belongs))*
THEN tEmpId $\mid\leftarrow$ *newEmpId;*
empRel $\mid\leftarrow$ *empRel* $\cup$ {*(tEmpId, name, belongs, works)*};
HireEmployee $\mid\leftarrow$ *tEmpId*
ELSE HireEmployee $\mid\leftarrow$ *nilEmpId*
END
END HireEmployee;
OTHER newEmpId $\in$ ($\rightarrow$ *(EmpIds - employees));*

Our example demonstrates that a refined operation may have a weaker precondition than the initial one: HireEmployee is now defined in all cases in which the static type predicate holds (a condition which can already be verified at compile time). It either performs the required state transition or it returns as an exception a specific value, nilEmpId.

4.4 Final Mapping into DBPL Programs

A language like DBPL, with a rich typing system and with first-order calculus expressions for set evaluation and control, turns out to be an appropriate target for the implementation of sufficiently refined Abstract Machines. Informally speaking, we can compile into equivalent DBPL programs those machines which have variables constrained by static type predicates and whose operations are defined by deterministic and sequential Generalized Substitutions.

In the final refinement step of HireEmployee, the "static" condition is transformed into DBPL parameter types, and the conditional substitution results in a database update statement controlled by a first-order database query expression (Appendix, Fig. 2).

5 Summary

We have presented, rather sketchily, some of the highlights of two languages, TDL and DBPL, for the conceptual design and the implementation of an information systems.

TDL is based on the language Taxis, but it has been refurbished to permit the predicative description of procedures (transactions, functions, script transitions), integrity constraints and coordinated transaction groups (scripts). TDL's expression language, unlike Taxis, is based on set theoretic notions. TDL maintains an object-oriented approach, thus allowing designers who map Telos requirements into TDL designs to concern themselves mostly with global issues such as view integration and the elimination of omnipresent temporal indices.

On the other hand, TDL adopts the standard (destructive) state based model of computation of most procedural languages. The mapping to DBPL programs can therefore concentrate on the appropriate choices for data identification, structuring and typing, as well as corresponding refinement steps on transactions and functions.

As argued in the previous section, Abrial's methodology of refining abstract machines appears to be a promising approach in which to study the issue of provably correct implementation. In this setting, a language like DBPL, with a set- and predicate-oriented approach to data modeling, a rich typing system, and abstract support for concurrency, recovery and persistence, turns out to be an appropriate target for the refinement of TDL designs into database application software.

Acknowledgment: We would like to thank our DAIDA colleagues, in particular M. Jarke, F. Matthes, and M. Mertikas, for their contributions to this research.

Appendix

```
TDLDESIGN ResearchCompanies IS
  ENUMERATED CLASS Agencies = {'ESPRIT, 'DFG, 'NSF, ...};
  ENTITY CLASS Companies WITH
    UNIQUE, UNCHANGING name : Strings;
    CHANGING engagedIn : SetOf Projects;
  END Companies;
  ENTITY CLASS Employees WITH
    UNCHANGING name : Strings;
    CHANGING belongsTo : Companies; worksOn : SetOf Projects;
    INVARIANTS onEmpProj: True IS (this.worksOn subsetOf this.belongsTo.engagedIn);
  END Employees;
  ENTITY CLASS Projects WITH
    UNIQUE, UNCHANGING name : Strings; getsGrantFrom : Agencies;
    CHANGING consortium : SetOf Companies;
    INVARIANTS onProjComp:
      True Is (this.consortium = {each x in Companies : this isIn x.engagedIn});
  END Projects;
  TRANSACTION HireEmployee WITH
    IN name : Strings; belongs : Companies; works : SetOf Project;
    OUT, PRODUCES e : Employees;
    GIVEN works subsetOf belongs.engagedIn;
    GOALS (e.name' = name) and (e.worksOn' = works) and (e.belongsTo' = belongs);
  END HireEmployee;
END ResearchCompanies;
```

Fig. 1: TDL Design of the ResearchCcompanies Example

```
DEFINITION MODULE ResearchCompaniesTypes;
  IMPORT Identifier,String;
  TYPE
    Agencies = (ESPRIT, DFG, NSF, ..);
    CompNames, EmpNames,ProjNames = String.Type;
    EmpIds = Identifier.Type;
    ProjIdRecType = RECORD name : ProjNames; getsGrantFrom : Agencies END;
    ProjIdRelType = RELATION OF ProjIdRecType;
    CompRelType = RELATION name OF
                  RECORD name : CompNames; engagedIn : ProjIdRelType END;
    EmpRelType = RELATION employee OF
                  RECORD employee : EmpIds; name : EmpNames;
                         belongsTo : CompNames; worksOn : ProjIdRelType END;
    ProjRelType = RELATION projId OF
                  RECORD projId : ProjIdRecType;
                         consortium : RELATION OF CompNames END;
END ResearchCompaniesTypes.

DEFINITION MODULE ResearchCompaniesOps;
  FROM ResearchCompaniesTypes IMPORT EmpNames, CompNames, ProjIdRelType, EmpIds;
  TRANSACTION HireEmployee(name:EmpNames;belongs:CompNames; works:ProjIdRelType) : EmpIds;
END ResearchCompaniesOps.
DATABASE IMPLEMENTATION MODULE ResearchCompaniesOps;
  FROM ResearchCompaniesTypes IMPORT CompRelType; EmpRelType; ProjRelType;
  IMPORT Identifier;
  VAR compRel : CompRelType;
      empRel : EmpRelType;
      projRel : ProjRelType;
  TRANSACTION HireEmployee (name:EmpNames;belongs:CompNames; works:ProjIdRelType) : EmpIds;
    VAR tEmpId : EmpIds;
    BEGIN
      IF SOME c IN compRel (c.name = belongs) AND
        ALL w IN works (SOME p IN compRel[belongs].engagedIn (w = p))
      THEN tEmpId := Identifier.New;
        empRel :+ EmpRelType{{tEmpId,name,belongs,works}};
        RETURN tEmpId
      ELSE RETURN Identifier.Nil
      END
    END HireEmployee;
END ResearchCompaniesOps.
```

Fig. 2: DBPL Implementation of the ResearchCompanies Example

References

[1] Abrial, J.R., *Formal Construction of an Order System*, 26 Rue des Plantes, Paris 75014, May 1988.

[2] Abrial, J.R., *Introduction to Set Notations*, Parts 1,2, 26 Rue des Plantes, Paris 75014, June 1988.

[3] Abrial, J.R., P. Gardiner, C. Morgan, and M. Spivey, *Abstract Machines*, Part1 - Part4, 26 Rue des Plantes, Paris 75014, June 1988.

[4] Abrial, J.R., C. Morgan, M. Spivey, and T.N. Vickers, *The Logic of 'B'* , 26 Rue des Plantes, Paris 75014, September 1988.

[5] Barron, J., *Dialogue and Process Design for Interactive Information Systems*, ACM SIGOA Conf. Proc., Philadelphia, June 1982.

[6] Borgida, A., *Language Features for Flexible Handling of Exceptions*, ACM Trans. on Database Systems 10(4), December 1985, pp.565-603.

[7] Borgida, A., M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou, *The Software Development Environment as a Knowledge Base Management System*, In: Schmidt, J.W., C. Thanos, (eds.), *Foundations of Knowledge Base Management*, Springer-Verlag, (in print).

[8] Borgida, A., S. Greenspan, and J. Mylopoulos, *Knowledge Representation as a Basis for Requirements*, IEEE Computer Vol.18, No.4, pp.82-103, April 1985.

[9] Borgida, A., J. Mylopoulos, and H.K.T. Wong, *Generalization/Specialization as a Basis for Software Specification*, In: Brodie, M.L., J. Mylopoulos, and J.W. Schmidt, (eds.), *On Conceptual Modelling, Springer-Verlag*, 1984.

[10] Borgida, A., J. Mylopoulos, J.W. Schmidt, and I. Wetzel, *Support for Data-Intensive Applications: Conceptual Design and Software Development*, Proc. 2nd Intl. Workshop on Database Programming Languages, Morgan Kaufmann, (to appear in 1989).

[11] Borgida, A., M. Koubarakis, J. Mylopoulos, and M. Stanley, *Telos: A Knowledge Representation Language for Requirements Modeling*, Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, February 1989.

[12] Chung, K.L., *An Extended Taxis Compiler*, M.Sc thesis, Dept. of Computer Science, University of Toronto, January 1984.

[13] DAIDA, *Final Version on TDL Design*, ESPRIT Project 892, DAIDA, Deliverable DES1.2, 1987.

[14] Guttag, J., J.J. Horning, and J.W. Wing, *Larch in five easy Pieces*, TR 5, DEC Systems Research Center, 1985.

[15] Hayes, I. (ed.), *Specification Case Studies*, Prentice Hall International, Englewood Cliffs NJ, 1987.

[16] Jones, C.B., *Systematic Software Development using VDM*, Prentice Hall, London, 1986.

[17] Matthes, F., A. Rudloff, and J.W. Schmidt, *Data- and Rule-Based Database Programming in DBPL*, Esprit Project 892, DAIDA, Deliverable IMP 3.b, March 1989.

[18] Mylopoulos, J., P. Bernstein, and H. Wong, *A Language Facility for Designing Data-Intensive Applications*, ACM TODS 5(2), June 1980.

[19] Nixon, B.A., K.L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, and M. Stanley, *Design of a Compiler for a Semantic Data Model*, In: Schmidt, J.W., C. Thanos, (eds.), *Foundations of Knowledge Base Management*, Springer-Verlag, (in print).

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

[21] Schmidt, J.W., H. Eckhardt, and F. Matthes, *DBPL Report*, DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[22] Schmidt, J.W., and M. Mall, *Abstraction Mechanisms for Database Programming*, ACM SIGPLAN Notices, 18:6, 1983, pp. 83-93.

[23] Spivey, J.M., *The Z Notation Reference Manual*, Prog. Res. Group, Oxford University, 1987.

[24] Stemple, D., and T. Sheard, *Specification and Verification of Abstract Database Types*, ACM PODS Conference, Waterloo, Ontario, April 1984, pp.248-257.

[25] Wirth, N., *Programming in Modula-2*, Springer-Verlag, 1983.

[26] Wordsworth, J.B., *A Z Development Method*, IBM UK Lab Ltd., Winchester, 1987.

ER APPROACH

8th International Conference on Entity-Relationship Approach

18-20 October 1989 | **Toronto, Canada** | **Park Plaza Hotel**

Presented by:

THE ER Institute

CONFERENCE COMMITTEE

US Conference Chairman
James P. Fry
University of Michigan

European Conference Chairman
Carlo Batini
Università di Roma

Program Committee Chairman
Frederick H. Lochovsky
University of Toronto

Steering Committee Vice-Chairman
Salvatore T. March
University of Minnesota

The Entity-Relationship Approach is the basis for many Database Design and System Development Methodologies. ER Conferences bring together practitioners and researchers to share new developments and issues related to the use of the ER Approach.

3-DAY CONFERENCE FEATURES

* Internationally Known Experts in Database and Information Management from both Academic and Practitioners Communities * Tutorials, Technical Presentations, Panel Discussions, Vendor Demonstrations * Conference Proceedings and Tutorial Materials

PROGRAM HIGHLIGHTS

Paper Sessions:
* Database Design Methodologies
* Database Design Tools
* Schema and Query Translation
* Knowledge-Based Systems
* ER Model Extensions
* Graphical Query and Design Interfaces
* ER Query Languages

Panels:
* User Experience with ER Modeling
* Toward More Power for Database Systems, Models, and Users—What Should We Be Doing?
* Do We Really Need Object-Oriented Data Models?
* Beyond SQL: Query Languages for the '90's

Tutorials:
* M. Modell
  Introduction to the ER Model
* T. Catarci, C. Batini
  Università di Roma
  Visual Query Languages
* Y. Ioannidis
  University of Wisconsin
  AI and Databases
* T.J. Teorey
  University of Michigan
  Distributed Database Design

* Program information: Fred Lochovsky, Dept. of Computer Science, University of Toronto, (416) 978-7441; fred@db.toronto.edu

CONFERENCE LOCATION

The ER Conference will be held at the Park Plaza Hotel, 4 Avenue Road, Toronto, Canada, M5R 2E8 (Tel.: (416) 924-5471; Fax: (416) 924-4933). Room rate for single or double occupancy is $115.00/night (in Canadian dollars and subject to local tax). Room rate guaranteed only until Sept. 20, 1989. Thereafter reservations accepted on a space-available basis, possibly at a higher rate.

The Park Plaza Hotel is in downtown Toronto next to fashionable Yorkville and Bloor Street shops. The Royal Ontario Museum, Art Gallery of Ontario, Casa Loma, Eaton's Centre, and Chinatown are within walking distance. The hotel is right on the subway line offering convenient access to attractions such as Harbourfront, the CN Tower, the SkyDome, and the Ontario Science Centre.

CONFERENCE FEES:

	Until Sept. 15	After Sept. 15
Regular	$ 350	$ 400
ACM/IEEE CS member	$ 300	$ 350
Fulltime Student (no banquet)	$ 25	$ 35
Additional banquet tickets ___ x $ 35	$_____	$_____
Total amount	$_____	$_____

REGRISTRATION

James P. Fry
Center for Information Technology Integrat
University of Michigan
Ann Arbor MI 48109-2016
(313) 998-7944; Fax : 998-7944
jpfry@um.cc.umich.edu

Name: _____ Organization: _____
Address: _____
City: _____ Prov./State: _____ PC/Zip: _____
Country: _____ Telephone: _____

Make check or bank draft payable in US funds to 8th ER Conference or wire 072401006 Account # 30-301975-4 First of America, Ann Arbor MI 48104 USA

IEEE Computer Society
1730 Massachusetts Avenue, N W
Washington, DC 20036-1903

Henry F. Korth
University of Texas
Taylor 2124 Dept of CS
Austin, TX 78712
USA